Cloud Storage Systems: From Bad Practice to Practical Attacks

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cloud Storage Systems: From Bad Practice to Practical Attacks

Master's Thesis

M. Haller

March 6, 2022

Advisors: Prof. Dr. Kenny Paterson, Matilda Backendal

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

Abstract

Cloud storage security gained significant importance in the last decades due to the vast amount of outsourced sensitive information. Increased privacy awareness has led more and more cloud operators to adopt end-to-end encryption, removing the necessity for customers to trust the providers for data confidentiality. We analyze the cryptographic design of Mega, a cloud storage provider storing over 1000 petabytes of data for more than 243 million users. This thesis contributes four severe attacks allowing a malicious service provider or man-in-the-middle adversary who compromises the TLS connection to break the confidentiality and integrity of user keys and files. We exploit the lack of ciphertext integrity of the encrypted and outsourced RSA private key and characteristics of RSA-CRT to perform a binary search for one prime factor of the RSA-2048 modulus and recover the secret key – with lattice-based optimizations – in 512 user login attempts. During a single login attempt, the second attack decrypts any key ciphertext and exploits key reuse and knowledge of the RSA key. Furthermore, the third attack allows an attacker to frame users by inserting new files indistinguishable from genuinely uploaded ones. Finally, the fourth attack contributes a new variant of Bleichenbacher's attack on PKCS#1 v1.5 adapted for Mega's custom padding scheme, which tolerates small unknown prefix values through a new guess-and-purge strategy. We discuss significant challenges introduced by Mega's massive scale for a fundamental redesign of their architecture and suggest short-term and long-term countermeasures. We generalize our findings, examine the reasons for flawed cryptography in large-scale applications, and advocate for a cloud storage standard to improve the security and transparency of cloud providers in practice.

Acknowledgement

I would like to thank Prof. Dr. Kenny Paterson for sharing his immense experience with me. His inputs were always very useful and added both breadth and depth to this project. Furthermore, I am just as grateful for the feedback and ideas contributed by Matilda Backendal. Her talent for asking the right questions often challenged me to a more thorough investigation, which improved this thesis substantially. Moreover, her background in provable security added a new dimension to this thesis.

I enjoyed every meeting with my supervisors, they were always an enthusiastic and insightful exchange of ideas. The entire Applied Cryptography Group was very welcoming and offered helpful advice and conversations on numerous occasions. Overall, this outstanding research experience reassured me in my decision to start a Ph.D. after finishing my Master's degree.

Contents

Contents **iii**

1 Introduction **1**
1.1 Related Work 2
1.2 Contributions 3

2 Background **5**
2.1 Notation 5
2.1.1 Object Oriented Syntax 6
2.2 Cryptographic Zoo 7
2.2.1 ZXCVBN 7
2.2.2 AES 7
2.2.3 RSA 9
2.2.4 Ed25519 9
2.2.5 Curve25519 9
2.2.6 X.509 10
2.2.7 PBKDF2 10
2.2.8 CSPRNG 11

3 Cloud Data Storage Solutions **13**
3.1 Cloud Storage Provider Overview 13
3.1.1 BoxCryptor 15
3.1.2 DropBox 15
3.1.3 Google Drive 16
3.1.4 Icedrive 16
3.1.5 iCloud 16
3.1.6 Keybase 17
3.1.7 Mega 17
3.1.8 Nextcloud 17
3.1.9 OneDrive 18

3.1.10 pCloud . . . 18
3.1.11 Seafile . . . 18
3.1.12 Sync . . . 19
3.1.13 Syncthing . . . 19
3.1.14 Tresorit . . . 19
3.2 Cloud Storage Provider Features . . . 19
3.3 Mega's Cryptographic Design . . . 20
3.3.1 Registration . . . 21
3.3.2 Client Authentication . . . 24
3.3.3 Node Encryption . . . 26
3.3.4 Node Sharing . . . 28
3.3.5 Account Recovery . . . 29
3.3.6 Business Users . . . 30
3.3.7 Source Code Observations . . . 30
3.3.8 Privacy . . . 32
3.3.9 Confidentiality . . . 32
3.3.10 Authentication . . . 34
3.3.11 Integrity. . . . 35

4 Attacks on Mega **37**
4.1 Mega RSA Key Recovery Attack . . . 37
4.1.1 Mega's RSA Encryption . . . 38
4.1.2 Threat Model . . . 39
4.1.3 Factoring the RSA Modulus Using Binary Search and Robustness Properties of RSA-CRT . . . 40
4.1.4 Reducing the Number of Queries . . . 43
4.1.5 Proof of Concept Attack . . . 46
4.2 Mega Plaintext Recovery for AES-ECB Under the Master Key 48
4.2.1 Threat Model and Preliminaries . . . 48
4.2.2 Attack Description . . . 49
4.3 Mega Framing Attack to Place a New File . . . 51
4.3.1 Threat Model . . . 52
4.3.2 Framing Attack Description . . . 52
4.3.3 Proof of Concept Attack . . . 54
4.3.4 Framing Attack Without Decryption Oracle . . . 57
4.3.5 Conclusion . . . 58
4.4 Bleichenbacher Variant With Unknown Prefix . . . 58
4.4.1 Oracle in Mega's Legacy Chat Key Decryption . . . 58
4.4.2 Threat Model . . . 59
4.4.3 Guess-And-Purge Variant of Bleichenbacher's Attack . 60
4.4.4 Optimizations . . . 64
4.4.5 Analysis of the Introduced Modifications . . . 65
4.4.6 Empirical Evaluation . . . 67
4.4.7 Conclusion . . . 70

4.5 Conclusion 72

5 Mitigation of Attacks on Mega **73**
5.1 Immediate Countermeasures 74
5.1.1 Ad Hoc Integrity Protection for Key Ciphertexts and File Attributes 74
5.1.2 Master Key Separation 76
5.1.3 Stricter RSA Padding Format 77
5.2 Minimal Countermeasures 78
5.2.1 AEAD for Key Ciphertexts 78
5.2.2 RSA-OAEP for RSA Sharing Keys 79
5.3 Recommended Measures 80
5.3.1 Refactoring Node Encryption 81
5.3.2 Augmented PAKE for Authentication 83
5.3.3 Recommended Extension 84
5.4 Conclusion 84

6 Discussion of the Status Quo and Future Work **87**
6.1 How Mega's Cryptographic Design Fails 87
6.2 The Consequences of Mega's Flawed Design 89
6.3 Why Mega's Cryptographic Design Fails 91
6.4 Future Work: Cloud Storage Standard 92
6.5 Conclusion 93

Bibliography **95**

A Glossary **111**

B Nextcloud's Cryptographic Design **115**
B.1 Architecture 115
B.1.1 Registration 116
B.1.2 Client Authentication 116
B.1.3 Node Encryption 116
B.1.4 Node Sharing 118
B.1.5 Account Recovery 120
B.2 Observations and Properties 121
B.2.1 Source Code Observations 123
B.2.2 Privacy 123
B.2.3 Confidentiality 124
B.2.4 Authentication 124

C Mega TLS Cipher Suites **125**

Chapter 1

Introduction

In the last decades, the popularity of cloud storage services increased significantly, and they are projected to store 50% of the global data – corresponding to 100 zettabytes – by 2025 [143]. The benefits of outsourcing data are sharing, convenient access from multiple devices, and automatic backups. However, storing private information online changes the threat model dramatically. Previously, physical presence or tricking users into disclosing their data (e.g., through social engineering or infected user devices) was necessary to compromise data. Today, a vulnerability in a major cloud service provider simultaneously affects millions of customers. Therefore, attacks on outsourced data can be remotely exploitable and easily applied on a grand scale. At the same time, the digitalization of our world increases, and more sensitive data – including scanned contracts, corporate secrets, and medical histories – are gathered and stored digitally. COVID-19 recently accelerated this already remarkable digitalization trend significantly [78]. The Snowden revelations increased public awareness of privacy issues and fostered the adoption of end-to-end encryption [109]. We examine the state of encryption and documentation of fourteen cloud providers and find that although some popular services still did not transition to E2EE, many operators advertise secure and privacy-preserving storage. Consequently, these progressive cloud providers adopt a new threat model where customers are no longer required to trust them for confidentiality. In this thesis, we utilize the stronger adversary capabilities implied by this thread model to perform attacks.

We analyze Mega, one of the major cloud providers offering E2EE, and find substantial issues in their cryptographic design. A malicious cloud provider or strong MitM attacker can compromise the confidentiality of various keys and consequently all user files and chat messages. Our attacks recover the `RSA` private key of any particular user after as few as 512 login attempts. Subsequent attacks can decrypt file keys or forge a file indistinguishable from a genuinely uploaded one in a single login attempt. Among other issues

with Mega's cryptographic design, our attacks exploit the lack of integrity protection of key ciphertexts. We propose various measures and highlight practical issues with backward compatibility and the massive scale of Mega that make rigorous mitigation challenging.

The remainder of the introduction positions this thesis among previous research (Section 1.1) and details our contributions (Section 1.2). Chapter 2 introduces notation and specifies cryptographic primitives used in the following chapters and should serve as a quick reference. Next, we examine the suitability of existing cloud services for third-party analysis in Chapter 3 and analyze the design of Mega in detail. Building on the insights of this analysis, we present four attacks in Chapter 4: an RSA private key recovery attack, an AES-ECB plaintext recovery, a framing attack, and a Bleichenbacher-esque RSA decryption. In Chapter 5, we propose short-term and long-term mitigations and highlight various feasibility issues of invasive mitigation approaches due to the scale of Mega. We conclude in Chapter 6 by extracting general insights on the state of cloud storage security – and other cryptographic designs used at a large scale – from the case of Mega.

1.1 Related Work

Cloud Storage Surveys. As cloud storage became popular, several papers [73, 165, 168] discussed possible architectures and specifications, focusing on practical issues including redundant storage, load balancing, and physical security. In contrast, we consider the cryptographic design of end-to-end encrypted cloud services. Kamara and Lauter survey architectures for clouds supporting searchable encryption [122]. Our overview focuses on existing systems, their classical confidentiality guarantees, and their suitability for third-part analysis. Other comparisons [89, 145, 166] evaluated different aspects, including performance and pricing. The review [160] from 2011 is closer to our survey, albeit making a broader comparison with fewer details on the cryptographic primitives. Moreover, cloud storage providers developed significantly in the last decade.

Mega. Mega's security white paper [62] describes their cryptographic design. Compared to it, our source code analysis resulted in more insights and technical details, especially on their custom AES-CCM implementation, the key obfuscation, the chunk-wise file encryption, and the RSA decryption.

Key Overwriting and Power Fault Attacks. The first key recovery attack on RSA and DSA using key overwriting of the outsourced and encrypted key material appeared in [127] in the context of OpenPGP. We present a key recovery attack that observes results from the victim's RSA-CRT decryption with tampered key material. While previous work attacked RSA signatures,

we focus on encryption where only partial instead of complete output is available. Furthermore, power fault attacks on RSA signatures [93, 135] inspired our attack on RSA-CRT. Unlike them, we tamper with the private key instead of inducing errors in single computations. In summary, our attack setting and the capabilities of our adversary differ from previous work: the cited attacks target RSA signatures in a known-plaintext setting and require access to the full signature. We use chosen plaintexts to factor the RSA modulus, where we can observe only slightly more than a sixth of the decryption. Moreover, we can only modify the private key ciphertext at a granularity of 128 bits instead of the octet-granularity of [127]. Furthermore, we introduce a plaintext recovery attack that builds on the RSA private key recovery and is specific to the analyzed architecture.

CBC-MAC Cryptanalysis. We present a framing attack exploiting that CBC-MAC allows the construction of a file that produces a target MAC tag if the key is known, and we can choose a single AES block in the plaintext. We are not aware of any previous work which analyzed CBC-MAC in this unusual cryptanalytic setting where the adversary knows the key but cannot choose the tag. Related work on authenticated encryption without key commitment constructs ciphertexts that preserve essential structures of two file formats [77, 99, 134]. The authors utilize the junk-tolerance of parsers such that the decryption of one file produces garbled but ignored data where the other file is stored. In contrast, we only choose a single junk AES plaintext block for one key and target CBC-MAC instead of AES-GCM, AES-GCM-SIV, or OCB3.

Bleichenbacher-Esque Attacks on PKCS#1 v1.5. We contribute an RSA decryption attack that is a novel variant of Bleichenbacher's attack on PKCS#1 v1.5 from 1998 [88]. Other instances of this attack [90, 117, 141, 152, 169] exploit different side-channel leakage to build padding oracles. Unlike them, we do not target PKCS#1 v1.5 padding but a custom scheme that includes an unknown prefix circumventing the straightforward adaption of Bleichenbacher's attack. To the best of our knowledge, this thesis is the first to introduce and evaluate a guess-and-purge strategy to account for these unknown prefix values in the padding oracle.

1.2 Contributions

This thesis makes the following contributions:

1. We evaluate fourteen cloud providers on their suitability for analysis.
2. We provide detailed pseudo-code for the authentication, encryption, and sharing algorithms to enable the abstract analysis of the design of

Mega and Nextcloud.

3. We introduce attacks on Mega considering a malicious cloud provider or an active MitM adversary who compromised the TLS connection.
   - 3.1. RSA key recovery: combines key overwriting with a chosen-plaintext attack to factor the RSA modulus in 512 login queries.
   - 3.2. AES-ECB decryption: recovers the plaintext of two AES blocks encrypted with AES-ECB under the master key. In Mega's architecture, this affects signing keys, asymmetric chat keys, and node encryption keys using an adaption of the RSA key recovery attack.
   - 3.3. Framing attack: uses the AES-ECB decryption to place a largely chosen file (except for one AES block) in a victim's cloud, which is indistinguishable from genuinely uploaded data.
   - 3.4. Novel Bleichenbacher variant: motivated by Mega's custom RSA padding, we provide a more generic description of Bleichenbacher's attack on PKCS#1 v1.5 that can tolerate small unknown prefix values.
4. We suggest practical and backwards compatible patches of Mega's implementation to protect against our attacks in the short and long term.
5. We generalize our findings and hypothesize why large-scale services suffer from flawed designs, what the consequences of a compromise are, and advocate for a cloud storage standard to improve the security and transparency of cloud providers.

Chapter 2

Background

Before we study cloud providers and discuss various attacks in technical detail, we introduce some notation (Section 2.1) and specify cryptographic primitives (Section 2.2). The goal of this chapter is to carefully develop a common understanding of the exact interpretation of any formulation. We encourage the reader to use the following sections as lookup material, since most definitions are not repeated later on.

2.1 Notation

We define and use the following conventions in this thesis:

- $x \leftarrow y$; set element x to the value y
- $x \leftarrow\!\!\!\$\ \mathcal{X}$; sample element x from the set $\mathcal{X}$ uniformly at random
- $r \xleftarrow{\$} f(x)$; indicates a randomized function f, which in this instances produces r on input x
- $[m]_k$; ciphertext representing the encryption of message m with the key k
- $[m]_k^\sigma$; signature of the message m using the key k
- $\{pk, r\}_{sk_T}$; certificate binding the public key pk to the identity r, signed by the trusted authority's secret key sk_T.
- $|s|$; number of characters of the string s
- $|b|_2$; number of bits of the binary value b
- $|b|_8$; number of bytes of the binary value b, specifically $|b|_8 = \lceil |b|_2 / 8 \rceil$
- $s_1 || s_2$; string concatenation, appending s_2 to s_1

- $l[a{:}b]$; take the slice $\{e_a, e_{a+1}, \ldots, e_b\}$ from the list $l \leftarrow \{e_1, e_2, \ldots, e_n\}$, where $1 \leq a \leq b \leq n$. We treat strings as lists of characters and byte strings as lists of bytes.
- $[a, b]$; for the set $\{a, a+1, \ldots, b\}$, where $a < b$. Variants with parentheses exclude the boundary, e.g., $(a, b] = \{a+1, a+2, \ldots, b\}$.
- $\{0,1\}^n$; for the set of all n-bit strings
- $0^n, 1^n$; as shortcuts for the n-bit strings $00\ldots0$ respectively $11\ldots1$
- $\lfloor X \rfloor_p$; represents the integer rest when dividing X by p. In other words, there exist $a \in \mathbb{Z}, r \in [0, p)$ such that $X = a \cdot p + r$, where $r = \lfloor X \rfloor_p$. We use this to distinguish the integer rest from elements in the ring $\mathbb{Z}_p$[1].
- $\lfloor X^{-1} \rfloor_p$; represents the integer value of the inverse of X modulo p, i.e., it corresponds to $a \in \mathbb{Z}$ such that $0 \leq a < p$ and $a \equiv_p X^{-1}$.
- $||v||_1$; the `L1` norm of the vector $v = [v_n, v_{n-1}, \ldots, v_0]^T$, i.e., $||v||_1 = \sum_{i=0}^{n} |v_i|$
- $||v||_2$; the `L2` norm of the vector $v = [v_n, v_{n-1}, \ldots, v_0]^T$, i.e., $||v||_2 = \sqrt{\sum_{i=0}^{n} (v_i)^2}$
- $\texttt{lpad}_N\,(i)$; translates the integer i to N bytes of big-endian byte encoding. If necessary, the value is left padded with zero bytes to reach N bytes.

2.1.1 Object Oriented Syntax

Occasionally, we use object oriented syntax in the mathematical description of algorithms to make relationships explicit.

For instance, we write the following:

- `server.fn`(arg); for the API call of the function `fn` with the argument(s) `arg` on the server.
- `keychain.Put`(x), `keychain.Get`(x); to specify storing/retrieving a value x in/from the on-device key chain.
- `obj.x`; for the variable x stored in the context of `obj`.

[1] To give an example for the subtle difference between $X_r \leftarrow \lfloor X \rfloor_p$ and $X_p \leftarrow X \mod p$ (i.e., $X_p \in \mathbb{Z}_p$), we consider the multiplication with an integer $\alpha \in \mathbb{Z}$. First, we note that $\alpha \cdot X_p$ is strictly speaking not defined, because the ring multiplication operation is only defined for two elements in $\mathbb{Z}_p$. However, if we consider $\alpha_p \leftarrow \alpha \mod p$, then $Y_p \leftarrow \alpha_p \cdot X_p$ is an element in $\mathbb{Z}_p$. In other words $0 \leq \bar{Y}_p < p$, where $\bar{Y}_p$ is the value of Y_p lifted to integers. On the contrary, $Y_r \leftarrow \alpha \cdot X_r$ is a well defined operation over integers without any implicit reduction modulo p. Consequently, we can have $\bar{Y}_p \neq Y_r$ (however, we always have $\bar{Y}_p \equiv_p Y_r$), which is relevant when we combine operands that were reduced by different moduli, like in the CRT.

2.2 Cryptographic Zoo

In the following, we specify the various algorithms used by cloud service providers or us later in this thesis.

2.2.1 ZXCVBN

`ZXCVBN` is a practical and efficient password strength estimator that was presented by Daniel Wheeler from DropBoxInc. at USENIX 2016 [163].

We only use the following function of `ZXCVBN`:

- $i \leftarrow$ `ZXCVBN.score`(pw); estimates the strength of the password pw and returns $i \in [0, 4]$, where 0 indicates a very weak secret and 4 a strong one.

2.2.2 AES

We use different modes of operation for `AES`. Let $B = 128$ be the block size and $\mathcal{K}$ the key space. We use the block cipher $F : \mathcal{K} \times \{0,1\}^B \rightarrow \{0,1\}^B$, with $F(k, x) \rightarrow y$ for a key k, a plaintext block x and a ciphertext block y. In the following, we use $m = m_0||m_1|| \ldots ||m_r$, where $|m_0| = |m_1| = \ldots = |m_r| = B$ for an arbitrary plaintext message of r blocks and $c = c_0||c_1|| \ldots ||c_s$ where $|c_0| = |c_1| = \ldots = |c_s| = B$ for a ciphertext of s blocks.

- Electronic Codebook (ECB)
  - $c \leftarrow$ `AES-ECB.Enc`(k, m); encrypts the message m using the key k to obtain the ciphertext c where $c_i \leftarrow F(k, m_i) \forall i \in [0, r]$ and $r = s$.
  - $m \leftarrow$ `AES-ECB.Dec`(k, c); decrypts the ciphertext c using the key k to obtain the plaintext m for $m_i \leftarrow F^{-1}(k, c_i) \forall i \in [0, r]$ and $r = s$.
- Cipher Block Chaining (CBC) [103]
  - $c \leftarrow$ `AES-CBC.Enc`(k, m, iv); encrypts the message m using the key k and the initialization vector iv to obtain the ciphertext c, where $c_0 \leftarrow iv$ and $c_{i+1} \leftarrow F(k, m_i \oplus c_i) \forall i \in [0, r]$. Note that the ciphertext has one additional block, i.e. $s = r + 1$.
  - $m \leftarrow$ `AES-CBC.Dec`(k, c); decrypts the ciphertext c using the key k to obtain the message m. c_0 is the initialization vector and $m_i \leftarrow F^{-1}(k, c_{i+1}) \oplus c_i \forall i \in [0, r]$.
- Galois Counter Mode (GCM)[2]

[2]This is a simplified description, refer to the proposal by David McGrew and John Viega [140] and and NIST SP 800-38D [101] for the full specification.

  - $c, \tau \leftarrow \texttt{AES-GCM.Enc}(k, m, n, ad)$; encrypts the message m with the associated data ad using the key k and nonce n to obtain the ciphertext c and the authentication tag τ. We have the key stream blocks $Y_i \leftarrow F(k, G(n, i))$ where $G(n, i)$ calculates the counter value for position i. The ciphertext blocks are $c_i = m_i \oplus Y_{i+1}$. For authentication, we have $\tau \leftarrow H(k, ad, c) \oplus Y_0$, where H is a hash aggregation of its inputs.
  - $m/\perp \leftarrow \texttt{AES-GCM.Dec}(k, n, \tau, c, ad)$; attempts to decrypt the ciphertext c with nonce n, tag τ, and associated data ad. If the authentication fails, this returns $\perp$, otherwise, the plaintext m. We again have the key stream blocks $Y_i \leftarrow F(k, G(n, i))$ and compute $m_i \leftarrow c_i \oplus Y_{i+1}$. Authentication fails if $\tau \neq H(k, ad, c) \oplus Y_0$.

- Counter with CBC-MAC Mode (CCM)[3]
  - $c, \tau \leftarrow \texttt{AES-CCM.Enc}(k, n, iv_{mac}, m, ad)$; encrypts the message m with the associated data ad, the AES-CTR nonce n, and the CBC-MAC IV iv_{mac} using the key k. This results in the ciphertext c with the tag τ. Let $Y_i \leftarrow F(k, G(n, i))$ be the key stream blocks, where $G(n, i)$ calculates the counter value for position i. Then, the ciphertext blocks are $c_i \leftarrow m_i \oplus Y_{i+1}$. For authentication, we compute the AES-CBC encryption of $E(ad)||m$ with IV iv_{mac}, where E does the length encoding and padding of ad. Let d be the last block of this AES-CBC encryption, then we have $\tau \leftarrow d \oplus Y_0$.
  - $m/\perp \leftarrow \texttt{AES-CCM.Dec}(k, n, iv_{mac}, \tau, c, ad)$; attempts to decrypt the ciphertext c with nonce n, CBC-MAC IV iv_{mac}, tag τ, and associated data ad. Let $Y_i \leftarrow F(k, G(n, i))$ again be the key stream blocks. Then, the authentication tag is recomputed on the plaintext as in the encryption and compared against $\tau \oplus Y_0$. If the tags are not equal, the authentication failed and we return $\perp$. Otherwise, we return the plaintext m for $m_i \leftarrow c_i \oplus Y_{i+1}$.

Furthermore, we use the CBC-MAC message authentication code to integrity protect a message m split into blocks as described above.

- $\tau \leftarrow \texttt{CBC-MAC.Tag}(k, m, iv)$; produces the authentication tag τ of the message m using the IV iv for the underlying AES-CBC. The tag τ is the last AES ciphertext block of $\texttt{AES-CBC.Enc}(k, m, iv)$. Usually, $iv \leftarrow 0^{128}$ is used for CBC-MAC.
- $\perp/\top \leftarrow \texttt{CBC-MAC.Vfy}(k, m, iv, \tau)$; verifies the tag τ by recomputing $\texttt{CBC-MAC.Tag}(k, m, iv)$ and comparing the output to τ. Returns failure ($\perp$) or success ($\top$).

[3] RFC 3610 by Whiting et al. [164] describes this mode in more details.

2.2.3 RSA

RSA is one of the first public key schemes and was proposed by Rivest, Shamir, and Adleman in 1977 [151]. This influential cryptosystem is still widely used in 2021. In the following, we define the abstract notation used in the remaining part of this thesis.

- $sk, pk \xleftarrow{\$} \texttt{RSA.Gen}(l)$; generates a public key pk and the corresponding private key sk with an l-bit modulus.
- $c \xleftarrow{\$} \texttt{RSA.Enc}(pk, m)$; encrypts the message m using the public key pk and producing the ciphertext c. Since pk is public knowledge, anyone can perform this encryption. Normally, a randomized padding is applied to m (see below).
- $m \leftarrow \texttt{RSA.Dec}(sk, c)$; decrypts the ciphertext c to obtain the message m using the secret key sk.

RSA-ECB-OAEPWithSHA-256AndMGF1Padding. This is a combination of RSA with Optimal Asymmetric Encryption Padding (OAEP). Such a randomized padding is needed to achieve IND-CPA security and prevent other attacks in practice. ECB is only in the name for compatibility reasons and does not have any function. Finally, OAEP is instantiated with the hash function SHA-256 and the Mask Generation Function MGF1 [131].

2.2.4 Ed25519

Bernstein et al. proposed Ed25519 as a high-performance EdDSA signature scheme [85]. It is built on Curve25519 (cf. Section 2.2.5) and uses SHA-512.

We use the following Ed25519 operations:

- $sk, pk \xleftarrow{\$} \texttt{Ed25519.Gen}()$; generates a secret signing key sk and the corresponding public key pk, where the finite field for EdDSA uses the (almost) 256-bit prime $q = 2^{255} - 19$.
- $\sigma \leftarrow \texttt{Ed25519.Sign}(sk, m)$; create the Ed25519 signature σ from the message m by using the secret key sk.
- $\bot/\top \leftarrow \texttt{Ed25519.Verify}(pk, \sigma)$; verify the signature σ with the Ed25519 public key pk and succeed ($\top$) or fail ($\bot$).

2.2.5 Curve25519

Curve25519 is a high-speed elliptic curve that is primarily used for Diffie-Hellman key exchange and provides 128-bit security. Bernstein et al. released this fast and accessible curve in 2006 [84].

On an abstract level, the functions are defined as:

- $sk, pk \xleftarrow{\$} \texttt{Curve25519.Gen}()$; generate a secret key $sk \in [1, n-1]$, where n is the curve generator's order, and a public key pk (which is a point on the curve).
- $x_{A,B} \leftarrow \texttt{Curve25519.DH}(sk_A, pk_B)$; A calculates the shared secret $x_{A,B}$ between two entities A and B, where sk_A is A's secret key and pk_B is the public key of B. The other party B can compute the shared secret similarly as $\texttt{Curve25519.DH}(sk_B, pk_A)$.

2.2.6 X.509

X509 is a standard from the International Telecommunication Union for certificates that bind identities to public keys [116]. Such certificates are ubiquitous in the Internet architecture. The root of trust (e.g., pre-installed certificates in browsers) stores certificates of some trusted organizations. These entities use the secret keys corresponding to the trusted public keys to sign certificates and bind other identities to key material (e.g., associating the key of a web server with a domain name, as used in HTTPS). By definition, we transitively trust a certificate whenever we can establish a chain of signed certificates leading to the root of trust.

We use the following notation:

- $csr \leftarrow \texttt{X509.CSR}(sk, pk, id)$; creates a Certificate Signing Request (CSR) to bind the public key pk to the identity id. This request is signed with the secret key sk.
- $cert \leftarrow \texttt{X509.CreateCert}(sk, csr)$; create a certificate from the CSR by singing the binding described in csr with the secret key sk.
- $\bot/\top \leftarrow \texttt{X509.Vfy}(cert, pk)$; verify the signature of certificate $cert$ using the (trusted) public key pk and return failure ($\bot$) or success ($\top$). The used public key is either in the root of trust or has a trusted certificate itself.

2.2.7 PBKDF2

The Password-Based Key Derivation Function 2 (PBKDF2) is described in the Public-Key Cryptography Standards (PKCS) #5 from RSA Laboratories [131]. PBKDF2 is a countermeasure against brute force attacks and repeatedly applies a pseudo-random function (PRF) to the password and a randomly selected salt. In the following, we use two instantiations of this primitive: PBKDF2-HMAC-SHA1 and PBKDF2-HMAC-SHA512. They use the different hash functions SHA-1 respectively SHA-512 for the PRF HMAC.

We write a call to PBKDF2 as follows (PBKDF2-HMAC-SHA1 is defined similarly):

- $k \leftarrow$ `PBKDF2-HMAC-SHA512`$(pw, s, \text{iter}=i, \text{len}=l)$; derive an l-bit key k from the password pw and salt s in i iterations.

2.2.8 CSPRNG

A Cryptographically Secure Pseudo-Random Number Generator provides random numbers suitable for cryptography. Libraries usually provide such a secure source of randomness, e.g., the `RNGCryptoServiceProvider` class of `.NET` [42].

We use the following notation:

- $k \xleftarrow{\$}$ `CSPRNG.random_bits`(l); select key k from $\{0,1\}^l$ in cryptographically secure pseudo-random manner

Chapter 3

Cloud Data Storage Solutions

We examine the current landscape of cloud services and provide a broad overview of the suitability of existing solutions for independent audits in Section 3.1. We summarize the features that cloud providers commonly aim to achieve in Section 3.2. Furthermore, we discuss in Section 3.3 Mega's cryptographic design in detail based on information from their white paper [62] and analyzing their client's source code. This in-depth analysis of Mega's architecture lays the foundation for our attacks presented in Chapter 4.

In addition, Appendix B reviews the completely open-source code and design of Nextcloud and compares it to Mega's architecture. We focus our analysis in this thesis on Mega, because they are more widely deployed and have better integrated end-to-end encryption. Furthermore, attacks by a malicious service provider are within the threat model since Mega cannot be self-hosted like Nextcloud. However, future work could build on our summary of Nextcloud's design and search attacks on their system.

3.1 Cloud Storage Provider Overview

We briefly discuss the data protection approach, availability of documentation and source code, as well as the popularity of the following data storage solutions: BoxCryptor, DropBox, Google Drive, Icedrive, Mega, Nextcloud, OneDrive, pCloud, Seafile, Sync, Syncthing, and Tresorit. Table 3.1 summarizes the content of this section.

First of all, we introduce the four criteria that we use to evaluate the providers.

Type. The examined services use the following two approaches:

- `T1`. The vast majority offer a classical cloud storage with a server component, usually hosted by the provider, and clients for different operating systems.

- T2. Two solutions – BoxCryptor and Tresorit – can be used as a encryption solution *in addition* to an untrusted cloud storage. This software encrypts data before uploading.

Available Documentation. We consider a service to have documentation if it provides a white paper (or something similar) with sufficient technical information on the design of their solution. A service has partial documentation, if some relevant technical details (such as the used encryption algorithms) and a high-level discussion is available. We do not consider source code to be documentation, as this is evaluated separately.

Available Source Code. Note that we avoid calling this category "Open Source", since we are only interested in the reviewability of the code of a service, irrespective of the license under which it is published. Services that only publish their client's code earn the label "Client" if the relevant cryptographic operations are performed on the client and can thus be reviewed. We consider the extensive use of open source libraries or out-dated source code as "Partial" availability.

Popularity. We use the number of users as a rough indicator for the popularity of a service. However, there are two caveats to keep in mind with this approach: first, some companies publish their user statistics very irregularly or not at all. Consequently, we are in the difficult situation to compare numbers from different years. This inconsistency is problematic since user bases do not grow linearly. Second, there is no industry standard according to which active users are measured. It is entirely the companies' decision to define after which period of inactivity a user is no longer counted. Furthermore, identifying users can be challenging itself: on the one hand, a single user might have multiple accounts. On the other hand, a single account might be shared by multiple users.

E2EE. We evaluate whether the service claims to offer End-to-End Encryption (E2EE), which we define as local encryption of all data on the client, where the key material is only accessible on the client's devices. This property is sometimes also referred to as client-side encryption or zero knowledge encryption.

ToS. In this point, we state whether the provider's Terms of Service (ToS) *explicitly* disallow the analysis of their products ("No") or do not regulate it ("Yes"). For instance, many forbid to recover their source code by reverse engineering, decompilation, or other means. Most disallowing ToS are formulated in a way to make sure that reverse engineering is illegal if this is possible in the local legislation. However, the situation varies from country to

	Type	Avail. Docs	Avail. Source Code	Popularity [10^6 users]	E2EE	ToS
BoxCryptor	T2	Yes	Partially	0.5	Yes	No
DropBox	T1	Partially	No	700	No	No
Google Drive	T1	Yes	Partially	1000	No	No
Icedrive	T1	Partially	No	0.15	Yes	Yes
iCloud	T1	Partially	No	850	No	No
Keybase	T1	Yes	Client	0.1	Yes	Yes
Mega	T1	Yes	Client	243	Yes	Yes
Nextcloud	T1	Yes	Yes	20	Yes	Yes
OneDrive	T1	Partially	No	500 - 1000	Yes	No
pCloud	T1	Partially	Client	12	Yes	No
Seafile	T1	No	Yes	1	Yes	Yes
Sync	T1	Yes	No	1.7	Yes	No
Syncthing	T1	Yes	Yes	0.067	No	Yes
Tresorit	T1, T2	Yes	No	0.025	Yes	No

Table 3.1: Documentation status of popular cloud storage providers. Type T1 is a standalone storage provider and type T2 is a file encryption solution that can be used in combination with a cloud provider. The content of this table is to the best of our knowledge the current status as of 2021.

country, and the legal situation is complicated: for instance, the EFF provides an overview on how legal regulation affects reverse engineering in the United States [8].

3.1.1 BoxCryptor

BoxCryptor provides a local data encryption solution that is compatible with various cloud storage providers. Since they do not offer a server component, their product inherently provides client-side encryption. Their documentation is available on their website [153]. Although BoxCryptor published code to decrypt a single file on March, 2020 [6], this is neither the complete source code nor the latest version. For instance, the encryption part is missing entirely. BoxCryptor's ToS forbid reverse engineering of their product [7]. They reported in 2021 to have 500,000 users [1].

3.1.2 DropBox

DropBox offers a very popular cloud storage that is used by over 700 million users in 2021 [16]. Although they have a security white paper [64] for their business solution, we could not find technical details on the encryption

(e.g., the key rotation). Moreover, there is no white paper for the individual users' storage service. They store the user's file encryption key material on their servers [64]. DropBox supports the open source community and has published internal tools [15], however, their cloud storage is proprietary. DropBox's ToS do not allow reverse engineering [13].

3.1.3 Google Drive

Google Drive is the most wide-spread cloud storage in our comparison: already in 2018, Google reported over one billion users [132]. They provide two security white papers [60, 61] as well as an online resource on their encryption in transit [59]. Google uses and develops a variety of open source tools [21] such as BoringSSL. However, their cloud service is not published. The keys for Google Drive's encryption at rest are managed centrally by Google [60]. Google disallows reverse engineering any of their products in the general ToS [22], which also apply to Google Drive [20].

3.1.4 Icedrive

Icedrive is a comparatively new cloud storage service that started in 2019. To the best of our knowledge, they have not yet published the number of active users. We estimate them to have roughly 150,000 users based on the number of installs on Android devices. We calculate this estimate for the set of reference cloud storage providers $S = \{Mega, DropBox, GoogleDrive\}$ as follows:

$$\frac{1}{|S|} \sum_{s \in \mathcal{S}} I_{Icedrive} / I_s \cdot U_s \tag{3.1}$$

where I_s is the estimated number of installs on Android devices from androidrank [14, 19, 24, 30] and U_s is the number of active users listed in table 3.1 for the service provider s.

Icedrive's websites explains their encryption on a fairly high-level[25]: they use Twofish for client-side encryption of files and meta-data. However, we did not find any further details on their architecture, key wrapping, or key rotation.

3.1.5 iCloud

Apple's closed source cloud storage service iCloud is automatically enabled on Apple's devices starting from iOS 9. In 2018, Barclays' analysts estimated iCloud to have 850 million users [148]. Some Apple services – including health and home data – use E2EE. However, other data – such as iCloud drive, backups, and photos – is only encrypted with `AES`-128 in transit and at

rest, with keys known to Apple [66]. The available documentation provides few technical details on the implementation. The ToS of iCloud [67] do not explicitly mention reverse engineering; however, they prohibit any copy of the software, which is often part of the reverse engineering process. The terms and conditions for Apple media services, which may be argued to include iCloud, explicitly prohibit any attempt to derive the source code.

3.1.6 Keybase

Keybase is a chat and document sharing service with over 100'000 registered users [27], which clearly describes their design and threat model [126]. They distinguish themselves from other services by enabling users to link their accounts to their online presence by posting signed statements on other public accounts. For authentication, a user defines a set of public profiles they know to belong to the target person, and Keybase verifies the publicly posted signatures on these accounts. Keybase realizes E2EE with device-specific public keys associated with every account. Interestingly, Keybase uses multiple Merkle trees to make server and user actions publicly verifiable and malicious behavior detectable. Keybase's unusual approach entails building a public graph linking social media and key material, which raises privacy concerns.

The continuation of Keybase is unclear since only the code of their clients [28] but not their servers [161] is open source. Therefore, only Zoom, which acquired Keybase in 2020 [167], can continue the project. However, this seems unlikely since the development activity of Keybase has decreased significantly after the ownership change [130]. They publish most client software under the BSD 3-Clause license, and Keybase's ToS do not mention reverse engineering.

3.1.7 Mega

Section 3.3 describes the cloud storage service Mega in detail. They provide a detailed white paper [62] and the source code of their clients [55]. However, the server code is proprietary, presumably for economic reasons. In 2021, they report to have over 243 million users [3]. Although Mega's ToS disallow analyzing their products in general, there is an exception for the clients published under open source licenses [71].

3.1.8 Nextcloud

The architecture of Nextcloud's storage service is described in two white papers: one on E2EE [146] and the other on their server-side encryption mode. They completely open source their clients and servers [35] and provide a manual for developers [32]. It is challenging to estimate the number of

Nextcloud users, since they are spread over self-hosted Nextcloud server instances. In 2017, Nextcloud estimated to have "well over 20 million users" spread on tens of thousands of servers [150]. Today, there are over 400'000 server instances [2] and we therefore expect the number of users to be multiple factors higher. Nextcloud explicitly welcomes the public scrutiny of their cryptographic design [146].

We describe Nextcloud in detail in Appendix B. Nextcloud's documentation [146, 147] includes limited technical details, and they have not yet implemented all of the described concepts. We derive detailed pseudo-code algorithms for registration and node encryption from the source code. Furthermore, we discuss plausible future implementations for sharing folders.

3.1.9 OneDrive

OneDrive has multiple resources where they discuss their encryption on a high-level [56, 72, 139]. However, some implementation choices of their closed source product remain unclear, especially concerning the non-business instances. Microsoft does not frequently publish the usage statistics of individual products. Back in 2015, OneDrive's CEO Satya Nadella disclosed that they have more than 500 million users [125]. In April 2021, they announced to have over 1.3 billion monthly active Windows 10 users. Since OneDrive is installed by default on all these devices, we expect the current number of OneDrive users to be around one billion too. Microsoft disallows any reverse engineering for all their services, including OneDrive, in their service agreement [69]. OneDrive Business uses per-file keys to protect data at rest, however, the keys are stored by Microsoft [72].

3.1.10 pCloud

pCloud offers a cloud storage with client-side encryption of individual folders [40]. They briefly describe pCloud's general architecture [40, 53], however, some information is missing. For instance, they do not describe how their Merkle-tree based authentication works. They publish multiple SDKs [37] and the code for at least some clients [41]. pCloud's ToS ban analyzing their proprietary services [39]. We decided to give them a "No" for the ToS column, because neither the server-side code nor the source code of their main clients is published. Consequently, a significant part of their services cannot be analyzed. In 2021, pCloud reports having over 12 million users [38].

3.1.11 Seafile

Seafile is a completely open source cloud storage [44, 45, 46] with client-side encryption [43]. However, apart from the code, there is little documenta-

tion about the general architecture. Seafile's ToS naturally do not prohibit analyzing their code [29]. They report to have one million users in 2021 [43].

3.1.12 Sync

Sync's end-to-end encrypted cloud storage architecture is described in a white paper from 2015 [156]. Although this specification tends to be a bit vague (e.g., regarding the implementation) and somewhat dated, it does give an overview about the used primitives. They do not publish their source code and their ToS disallow reversing [52]. Sync reports to have 1.7 million users in 2021.

3.1.13 Syncthing

Syncthing provides peer to peer file synchronization without particular focus on security [47]. Documentation about the general architecture is rare. From their transparent usage reports, we can see that 67,000 users are using Syncthing in 2021 [51]. As a decentralized open source project [50], they do not have ToS.

3.1.14 Tresorit

Tresorit provides a solution that can either be used as standalone E2EE cloud service or as encryption layer for another untrusted service. Their architecture is described in two white papers [158, 159]. Although they specify which third party tools they use [54], some of which are open source, their own product is proprietary. We were only able to find that they had 25,000 business users in 2019 [157], but not the total number (which includes the individual users). Tresorit defines in their Acceptable Use Policy [4], which is part of their ToS [12], that reversing their products is not allowed.

3.2 Cloud Storage Provider Features

Based on our survey, we identify the following advanced features to be of particular relevance for an E2EE cloud storage service. Although by far not all of the studied providers achieves all of the mentioned properties, they seem desirable in general.

- *Multi-device support*; a user can access and manage his files from different devices
- *File sharing*; nodes can be made accessible to other users or with public/password-protected links
- *Share management*; users can manage the people with whom they share a folder and securely remove individuals

- *Account Recovery*; users who lost their key material can regain access to their encrypted data
- *File Versioning*; files keep a history of their modifications
- *Contact Relationships*; users can establish trusted connections with other entities and verify their identity and key material. Ideally, these relationships are private to prevent malicious providers from gathering metadata.

3.3 Mega's Cryptographic Design

We chose to take a closer look at Mega because they have a large user base, good documentation, and publish the source code of their clients. Furthermore, they focus on security and provide end-to-end encryption.

Kim Dotcom, Mathias Ortmann, and Bram van der Kolk launched Mega in 2013 and claimed it to be the "only major cloud storage provider supporting browser access to end-to-end encrypted cloud storage" [62]. Initially, Mega received some pushback for their system design. Prof. Alan Woodward pointed out two issues of doing cryptography in JavaScript [95]: first, the encryption code is loaded every time the user visits Mega's website. Therefore, a malicious administrator or a MitM could perform targeted attacks and replace the served encryption code with an instance that does not properly encrypt the files or discloses the keys. Second, the random numbers generated in JavaScript that Mega uses for key material were known to be weak. Furthermore, there were concerns that Mega would be used predominately to share copyright infringing material as its predecessor Megaupload [119]. Since Dotcom's previous website was shut down by law enforcement, it was unclear whether authorities could force Mega to adapt their service and disclose customers' keys [95]. In 2015, Kim Dotcom caused further controversy by announcing that he no longer trusts Mega and is not involved anymore [26], his claims being repudiated by Mega [58]. A company based in Hong Kong now holds 99.8% of Mega Limited's shares [31].

Despite all initial controversy, Mega continues to grow rapidly and now has over 10 million daily active users [3] and stores more than 1000 PB of data [65]. Mega's security white paper addresses some of the initial concerns and discusses their design decisions [62]. The service employs a rather stringent takedown policy and quickly removes public links to material violating copyright. They report to have received 746,336 takedown requests in Q3 2021, which corresponds to only 0.0007% percent of the total number of files stored on Mega [68]. Furthermore, they offer various alternative clients (desktop, Android, iOS) to their web browser. Additionally, there is a browser extension that avoids re-downloading the encryption-performing code. Nevertheless, normal web browser users still get the JavaScript code

served on connection. Concerning the insecure randomness, browsers have improved significantly and now provide CSPRNG. Section 3.3.7 discusses the last two points in more detail.

In this case study, we first present Mega's implementation for registering users (Section 3.3.1), authenticating clients (Section 3.3.2), encrypting files and folders (Section 3.3.3), and sharing (Section 3.3.4). Furthermore, we briefly discuss the account recovery (Section 3.3.5) and business user (Section 3.3.6) features. Finally, we discuss the more higher-level properties of privacy (Section 3.3.8), confidentiality (Section 3.3.9), authentication (Section 3.3.10), and integrity (Section 3.3.11).

Algorithm 1 Mega's user registration function for the email *email*, the password p_1, and the verification password p_2

1: **procedure** REGISTER_USER($email$, p_1, p_2)
    ▷ Email verification
2: **if** $|email| > 190$ **then**
3: **return false**

    ▷ Password verification
4: **if** $p_1 \neq p_2$ **or** $|p_1| < 8$ **or** `ZXCVBN.score`$(p_1) = 0$ **then**
5: **return false**

    ▷ Create key
6: $N_C, [k_M]_{k_e}, h_{k_a} \xleftarrow{\$}$ `MASTER_KEY_GEN`(p_1)

    ▷ Create user
7: `server.create_user`$(name, surname, email, N_C, [k_M]_{k_e}, h_{k_a})$

3.3.1 Registration

Algorithm 1 gives an overview of Mega's user registration. The new customer needs to enter an email address *email* and a password p_1. Following common practice, p_2 is the repetition of the latter for verification. The email address has to be at most 190 characters, because it is later used in a hash function with constant bounded input size (to thwart timing side-channels). Passwords that are too short or scored by `ZXCVBN` to be too weak are not accepted.

Next, the client generates the user's master key (cf. Algorithm 2) and calls the server's API to register the user. We remark that the generated master key does not depend on any user identifier. Mega justifies this decision in their security white paper [62] with usability: the only identifier on Mega is the user's email address. If the master key would depend on this contact

information, changing it would make all the user's data inaccessible unless it is re-encrypted. It is unclear, why they are not using a user ID, which is detached from the email. However, equal passwords of different users do not lead to the same master key as there are no collisions in the client nonces. Since the nonce is 128 bits, the Birthday Paradox tells us that approximately 2^{64} user accounts with the same password would be required to find a collision.

Algorithm 2 Mega's master key generation function from the password pw

1: **procedure** MASTER_KEY_GEN(pw)
▷ Calculate the derived key k_d
2: $N_C \xleftarrow{\$}$ `CSPRNG.random_bits`(128) ▷ Client random value
3: $s \leftarrow$ `SHA-256`(`PAD`($'mega.nz'$)$||N_C$) ▷ Salt
4: $k_d \leftarrow$ `PBKDF2-HMAC-SHA512`($pw, s,$ iter=100'000, len=256)

▷ Calculate the derived encryption resp. authentication keys k_e and k_a
5: $k_e||k_a \leftarrow k_d$ ▷ $|k_e|_2 = |k_a|_2 = 128$

▷ Generate and encrypt master key k_M
6: $k_M \xleftarrow{\$}$ `CSPRNG.random_bits`(128)
7: $[k_M]_{k_e} \leftarrow$ `AES-ECB.Enc`(k_e, k_M)

▷ Calculate the hashed authentication key h_{k_a}
8: $h_{k_a} \leftarrow$ `SHA-256`(k_a)[:16] ▷ h_{k_a} is the leftmost 16B of the hash

9: **return** N_C, $[k_M]_{k_e}$, h_{k_a}

Algorithm 3 Mega's padding of the string s

1: **procedure** PAD(s)
2: **while** $|s| < 200$ **do**
3: $s \leftarrow s||'P'$
4: **return** s

Master Key Generation. Algorithm 2 describes the client's master key derivation. First, the client chooses a random 128-bit nonce N_C. The salt s is the `SHA-256` hash of N_C concatenated to a constant string padded with Algorithm 3. Next, it calculates the 256-bit derived key k_d from the password and s using `PBKDF2-HMAC-SHA512`. It splits this key into an encryption key k_e and an authentication key k_a, both consisting of 128 bits. Finally, it generates the 128-bit master key k_m using a CSPRNG and encrypts it with k_e using

AES-ECB. The key generation then returns the nonce, the encrypted master key, and the first 128 bits of the SHA-256 hash of the authentication key k_a. Note that the salt is not returned and ultimately not stored on the server. This is done to avoid timing side-channels: Mega calculates the salt for login requests with valid and invalid email addresses (cf. Algorithm 6).

User Creation. When the server receives a `create_user` request, it generates a random 128-bit token t and constructs the confirmation link:

$$'https://mega.nz/\#confirm'||\texttt{base64\_urlencode}('ConfirmCodeV2'||t||email)$$

The client sends this token back to the server's API. When it is correct, the user can log in with his password.

Additional Key Generation. On the very first login, right after confirming the email address, additional key material is generated. Algorithm 4 shows the generation of three public key pairs: an Ed25519 key pair for signatures, an RSA key pair for sharing data, and a Curve25519 key pair for messaging. The Ed25519 private key acts as root of trust and is used to sign the public keys of the latter two. These signatures, together with the public keys, are stored on the server. Moreover, the secret keys are encrypted by the master key k_m and also uploaded.

Algorithm 4 Mega's function to generate additional key material

1: **procedure** ADDITIONAL_KEY_GEN(k_m)
  ▷ Generate signature key pair
2: $sk_{sign}, pk_{sign} \xleftarrow{\$} \texttt{Ed25519.Gen}()$
3: $[sk_{sign}]_{k_M} \leftarrow \texttt{AES-ECB.Enc}(k_M, sk_{sign})$
  ▷ Generate public key pair for data sharing
4: $sk_{share}, pk_{share} \xleftarrow{\$} \texttt{RSA.Gen}(2048)$
5: $[sk_{share}]_{k_M} \leftarrow \texttt{AES-ECB.Enc}(k_M, sk_{share})$
6: $[pk_{share}]^{\sigma}_{sk_{sign}} \leftarrow \texttt{Ed25519.Sign}(sk_{sign}, pk_{share})$
  ▷ Generate chat key material
7: $sk_{chat}, pk_{chat} \xleftarrow{\$} \texttt{Curve25519.Gen}()$
8: $[sk_{chat}]_{k_M} \leftarrow \texttt{AES-ECB.Enc}(k_M, sk_{chat})$
9: $[pk_{chat}]^{\sigma}_{sk_{sign}} \leftarrow \texttt{Ed25519.Sign}(sk_{sign}, pk_{share})$
10: $\texttt{server.store}(([sk_{sign}]_{k_M}, pk_{sign}), ([sk_{share}]_{k_M}, pk_{share}, [pk_{share}]^{\sigma}_{sk_{sign}}),$
$([sk_{chat}]_{k_M}, pk_{chat}, [pk_{chat}]^{\sigma}_{sk_{sign}}))$

After the described initial setup actions, the user now has an account registered with Mega. The service provider stores various key material, where confidential parts are protected by the master key k_m, which is itself stored and encrypted by k_e. The latter is derived from the user's password and therefore device-independent.

3.3.2 Client Authentication

Algorithm 5 shows the client-side login procedure. First, we request the salt for a given *email* from the server. Despite not having access to the server's source code, Algorithm 6 shows how the white paper [62] describes the server functionality.

Algorithm 5 Mega's client login procedure for a given email *email* and password *pw*

1: **procedure** CLIENT_LOGIN(*email*, *pw*)
2: $s \leftarrow$ `server.get_salt`$(email)$
3: $k_d \leftarrow$ `PBKDF2-HMAC-SHA512`$(pw, s, \text{iter=}100'000, \text{len=}256)$
4: $k_e || k_a \leftarrow k_d$ ▷ $|k_e|_2 = |k_a|_2 = 128$
5: $[k_M]_{k_e}, [sk_{share}]_{k_M}, [sid]_{pk_{share}} \leftarrow$ `server.authenticate`$(email, k_a)$
6: $k_M \leftarrow$ `AES-ECB.Dec`$(k_e, [k_M]_{k_e})$
7: $sk_{share} \leftarrow$ `AES-ECB.Dec`$(k_M, [sk_{share}]_{k_M})$
8: $sid \leftarrow$ `RSA.Dec`$(sk_{share}, [sid]_{pk_{share}})$

Fetching the Salt. Recall from Algorithm 1 that the server stores the client nonce N_C. If the email address is in the database, the server fetches this nonce and recomputes the salt. Otherwise, the server uses N_S, a 128-bit random value chosen by the server once a year, to compute the salt. In an effort to prevent timing side-channels, both hashed strings are padded to 200 characters. Furthermore, the server adds a random delay for non-existant email addresses.

It depends on the server implementation whether this is an adequate measure to prevent leaking the email registration status. Let X, Y, Z be random variables. $X \in [0, 50]$ models the database accesses since Mega reports searches to take at most 50 ms [62]. Assuming that the database searches are linear and only depend on the position of the email address in the records, then X is uniformly distributed (over all registered addresses), and we have $\mathbb{E}[X] = 25$. Furthermore, $Y \in [0, 50]$ is the random variable for the chosen delay. Finally, Z represents the time needed for detecting a record's presence in the database. We first treat Z as independent of X, because this is implementation dependent: for instance, one could use Bloom filters for probabilistic keys presence detection and only traverse all records for

hits in the Bloom filter. In conclusion, the two branches of Algorithm 6 have the following expected delays: $\mathbb{E}[Z] + \mathbb{E}[X]$ if the email is present and $\mathbb{E}[Z] + \mathbb{E}[Y]$ otherwise. If Y is chosen uniformly at random, these expected values are the same over all email addresses.

However, there are two subtle issues with the previous analysis: first, for a given *fixed* email address *email*, we have a fixed search delay $x_{email} \in X$ if it is in the database. Otherwise, unless the chosen delay depends deterministically on *email*, Y is independent of *email*. Therefore, we have $\mathbb{E}[Z] + x_{email} \neq \mathbb{E}[Z] + \mathbb{E}[Y]$. If Y is chosen uniformly at random, this case can be detected by having a mean delay of $\mathbb{E}[Z] + 25$ and higher variance than the $x_{email} = 25$ case. In other words, the measured timing of multiple requests for the same email address would change more if the entry is not present. Second, it depends on the implementation if detecting the presence of an email address and searching its record are actually two processes. For instance, the straightforward approach always searches the database and defines no matching record as a non-existant entry. However, in that scenario, Z is not a random variable. We either have $\mathbb{E}[Y]$ if the entry is present, or $50 + \mathbb{E}[Y]$ if it is not present (since we always search the entire database in the latter case). Thus, a non-present email address could be detected by having longer time measurements. Both of these issues show potential timing side-channels depending on the unknown server implementation. Further research could test this in practice and try to overcome the challenges of network-induced noise and potential rate limiting. We further discuss the implications of leaking registered email addresses in Section 3.3.8.

Algorithm 6 Mega's server login part 1 (getting the salt)

```
1: procedure GET_SALT(email)
2:     if email in database DB then
3:         N_C ← DB.get(email)
4:         s ← SHA-256(PAD('mega.nz')||N_C)
5:     else
6:         s ← SHA-256(PAD(email||'mega.nz')||N_S)
7:         sleep(r), for r ←$ [0,50] miliseconds
8:     return s
```

Authentication The client proceeds to calculate the derived key k_d from the user password *pw* and the fetched salt *s* using `PBKDF2-HMAC-SHA512`. Next, it sends the rightmost 128 bits of k_d, the so called authentication key k_a, to Mega. The server hashes k_a with `SHA-256` and compares the leftmost 128 bits of the hash to the stored h_{k_a}. If these values are equal, authentication succeeds. The server generates a session ID *sid* and encrypts it with the

user's public key and returns it to the client, together with the encrypted master key and secret sharing key.

Mega does not mention whether the latency of this second database access is obfuscated. However, they use double HMAC to compare $h^{auth}_{k_a}$ and h_{k_a}. Instead of directly comparing the hashes, this technique hashes them again with a randomly chosen key and then compares the results [79]. This breaks the comparison's dependency on the length of the matching prefix, as optimized implementations terminate early on the first character that differs. Without this measure, an attacker could recover $h^{auth}_{k_a}$ character by character assuming the timing side-channel is sufficiently precise.

Algorithm 7 Mega's server login part 2 (authentication)

procedure AUTHENTICATE($email$, k_a)
  $h^{auth}_{k_a} \leftarrow \texttt{SHA-256}(k_a)$
  $h_{k_a}, [k_M]_{k_e}, ([sk_{share}]_{k_M}, pk_{share}) \leftarrow \texttt{DB.get}(email)$
  if $h^{auth}_{k_a}[:16] = h_{k_a}$ **then** ▷ Double HMAC comparison
    $sid \leftarrow_{\$} \mathcal{SID}$ ▷ $\mathcal{SID}$ is the set of session IDs
    $[sid]_{pk_{share}} \xleftarrow{\$} \texttt{RSA.Enc}(pk_{share}, sid)$
    return $[k_M]_{k_e}$, $[sk_{share}]_{k_M}$, $[sid]_{pk_{share}}$
  else
    return $\perp$

After authenticating, the client sends the session identifier *sid* with all its requests to prove its identity to the server.

3.3.3 Node Encryption

Algorithm 8 describes how Mega encrypts a file or folder, which is a file or folder in their terminology. They maintain a flat hierarchy, where each node has an (encrypted) handle pointing to its parent. As folders do not have content, only their attributes are encrypted. First, the client samples a random 128-bit node key k_F and a 64-bit nonce N_F. Large nodes are partitioned into chunks c_i of size between 128 KB and 1 MB (depending on the client) [48]. Let l_c be the number of AES blocks per chunk, which is between 2^{10} and 2^{13} since AES blocks are 128 bits. These chunks are then encrypted with Mega's custom AES-CCM implementation with key k_F. Algorithm 11 describes AES_CCM_ENC_MEGA in detail: it encrypts the counter blocks with AES-CTR to produce the key stream. Each counter consists of the 128-bit concatenation of the random file nonce N_F (leftmost 64 bits) and the AES block index (interpreted as 8 bytes in big-endian encoding). The concatenation in the counter blocks ensures that all AES blocks across different chunks are XORed with a key stream derived from unique and consecutive counter values. The

IV for the MAC of the plaintext is implicit in Mega's version of AES-CCM and set to $N_F||N_F$.

The client aggregates the file chunk MACs into a single condensed tag value M_{cond}. This corresponds to the CBC-MAC of $M_0||M_1||\ldots||M_{n-1}$, the concatenation of all chunk MACs with the key k_F. Currently, Mega only checks the integrity of a file by recomputing M_{cond} after downloading all chunks. In the future, they want to verify the integrity of individual chunks separately for performance reasons [48].

Algorithm 8 Mega's procedure to encrypt node F with the master key k_M

1: **procedure** NODE_ENC(k_M, F)
2: $k_F \xleftarrow{\$} \{0,1\}^{128}$ ▷ Node key
3: $N_F \xleftarrow{\$} \{0,1\}^{64}$ ▷ Node nonce

▷ Chunk-wise file encryption
4: $l_c \leftarrow 2^j$ for $j \in [10, 13]$ ▷ Number of AES blocks per chunk
5: $c_1||c_2||\ldots||c_n \leftarrow F$ ▷ Where $\forall i \in [1,n]\,.\,|c_i|_2 = 128 \cdot l_c$
6: $\forall i \in [1,n]\,.[c_i]_{k_F}, M_i \leftarrow \texttt{AES\_CCM\_ENC\_MEGA}(k_F, N_F||(i \cdot l_c), c_i)$

▷ Create condensed CBC-MAC M_{cond}
7: $M_{cond} \leftarrow 0^{128}$
8: **for** $i \in [1,n]$ **do**
9: $M_{cond} \leftarrow \texttt{AES-ECB.Enc}(k_F, M_{cond} \oplus M_i)$

▷ Obfuscated file key k_F^{obf}
10: $k_F^{obf} \leftarrow \texttt{OBFUSCATE\_FILE\_KEY}(k_F, N_F, M_{cond})$
11: $[k_F^{obf}]_{k_M} \leftarrow \texttt{AES-ECB.Enc}(k_M, k_F^{obf})$
12: $\texttt{server.store}([k_F^{obf}]_{k_M})$

▷ Upload encrypted file and attributes
13: $[attr]_{k_F} \leftarrow \texttt{AES-CBC.Enc}(k_F, attr, 0^{128})$
14: $\texttt{server.store}([attr]_{k_F})$
15: $\texttt{server.store}(\forall i \in [1,n].[c_i]_{k_F})$

Before encrypting the key k_F and sending it to the server, it is obfuscated together with M_{cond} and the 64-bit AES-CTR nonce N_F as described in Algorithm 9. The obfuscation aggregates the condensed MAC M_{cond} to the so-called metamac by splitting M_{cond} into four 32-bit chunks and XORing together the first two chunks as well as the last two chunks. The obfuscation intertwines the key k_F with the concatenation of the nonce N_F and the metamac M_{meta} resulting in the obfuscated file key k_F^{obf}. In addition to the

scrambled file key k_F^{scr}, the obfuscated file key has N_F and M_{meta} appended. This entire key is then encrypted with `AES-ECB`, i.e., also the nonce and `metamac` are hidden from the cloud provider. Unfortunately, Mega provides no reasoning for the design of Algorithm 9. We think their goal is to create some binding between the randomly selected key k_F, the nonce N_F, and MAC M_{cond} that are used to encrypt node F. If a single bit flips in the obfuscated key, the file key k_F changes due to the XOR. Consequently, the file decrypts to garbled chunks and the integrity validation fails with high probability.

Finally, the client encrypts the node attributes (including its type, file name, and parent handle) with `AES-CBC` and uploads this, together with the encrypted chunks, to the server. Although the IV is fixed to the zero vector, deterministic encryption is not a problem since every file has a different file key. However, we remark that the attributes have no integrity protection.

Algorithm 9 Mega's obfuscation of the file encryption key k_F and the condensed mac M_{cond}

1: **procedure** OBFUSCATE_FILE_KEY(k_F, N_F, M_{cond})
   ▷ Calculate `metamac` M_{meta}
2: $M_{cond}^0||M_{cond}^1||M_{cond}^2||M_{cond}^3 \leftarrow M_{cond}$ ▷ $|M_{cond}^i|_2 = 32, \forall i \in [0,3]$
3: $M_{meta} \leftarrow (M_{cond}^0 \oplus M_{cond}^1)||(M_{cond}^2 \oplus M_{cond}^3)$
4: $k_F^{scr} \leftarrow k_F \oplus (N_F||M_{meta})$
5: **return** $k_F^{scr}||N_F||M_{meta}$

3.3.4 Node Sharing

Mega supports different types of sharing: public links, password-protected links, and sharing with contacts. We briefly discuss all of them and go into more details on the second one.

Public Links. If you share a file publicly, the client generates an URL that contains the obfuscated file key. Since it is located after a hashtag, this value is not submitted to the server but only processed locally, in the receiver's browser. Therefore, the cloud provider does not get access to the public link and consequently does not see the key material required to decrypt the shared file. The Mega web app uses this value to decrypt the file. Sharing folders requires the extra step of generating a share key, which then encrypts all obfuscated node keys for files located in that folder. The public link then contains this share key.

Password-Protected Links. Algorithm 10 shows how the link share password is integrated with the existing node key (for files) or share key (for

folders). First, the client calculates a 512-bit derived key k_d from the share password pw and a randomly generated 256-bit salt s. The leftmost 16B (resp. 32B) of this key are simply XORed with the file key (share key). The rightmost 16B are used as MAC key k_{MAC}. Next, the client concatenates the 1B algorithm identifier (currently unused), a 1B node type (0 for a folder, 1 for a file), the 6B public file/folder handle, the salt, and the encrypted node key. The final link then contains this concatenation together with the $HMAC\text{-}SHA$-256 tag over it.

Algorithm 10 Mega's procedure to share a link to node F protected with the password pw

1: **procedure** SHARE_NODE(F, pw)
2: $s \leftarrow_{\$} \{0,1\}^{256}$ ▷ Salt
3: $k_d \leftarrow$ PBKDF2-HMAC-SHA512$(pw, s, \text{iter=}100'000, \text{len=}512)$

▷ Encrypt node key
4: $[k_F]_{k_d} \leftarrow k_F \oplus k_d[0{:}x]$, where $x = 16$ (file) or $x = 32$ (folder).

▷ Construct and MAC link
5: $k_{MAC} \leftarrow k_d[32{:}64]$
6: $l \leftarrow A||T||H||s||[k_F]_{k_d}$ ▷ Alg A, type T, public handle H
7: $m_{auth} \leftarrow HMAC\text{-}SHA\text{-}256(k_{MAC}, l)$
8: **return** $l||m_{auth}$

Sharing with Contacts Mega allows users to establish contact relationships inside their service. Users can send contact requests using the other's email address or scanning his or her QR code. The client stores the fingerprint of this key and warns the user if this value changes. Shared data for pending requests is not encrypted for the recipient until the contact relationship was established. To share a file with a contact, the sender encrypts the file key with the receiver's public RSA key (cf. Algorithm 4).

3.3.5 Account Recovery

Mega allows users to export their master key k_M and use it to reset an account password. For this purpose, the user sets a new password and re-encrypts the master key.

The mega client detects the correct master key by decrypting the RSA private key and verifying that it matches the user's public key. Since this is only a client-side check, a malicious user with a reset link can garble another user's data by resetting the password with an invalid recovery key.

Mega allows password resets without entering the old password in active sessions. They argue in the white paper [62] that many users who lost their password still have an active session on one of their devices. Therefore, this implementation allows such users to recover their accounts without knowing the master key. Technically, this is possible because the logged-in device still stores the master key. On the downside, this implies that if you can hijack a session, you can lock out the real user and gain exclusive access to his or her data by changing the password.

3.3.6 Business Users

Another functionality that Mega provides are business accounts. They are administered by master users, who have management permissions on the business account and can view the key material of sub-users. The master key of normal users is additionally encrypted with the business account's key, which in turn is encrypted for every master user. Clients should display when a business sub-user is created, so that people that pay attention cannot be tricked to share their key material with a malicious business owner.

3.3.7 Source Code Observations

The previous sections discussed Mega functionality on an abstract level. In the next part, we discuss some observations on the source code of Mega's SDK and web client [55].

Crypto++. Mega's SDK uses the open-source C++ library Crypto++ [9] for cryptographic primitives such as `AES-ECB` and hash functions. This library is still actively maintained and does not have any critical CVEs for the primitives used by Mega [11]. However, Mega implements some block cipher modes manually (see Section 3.3.9).

SJCL. The web client uses the Standford JavaScript Crypto Library (`SJCL`) contributed by Stark, Hamburg, and Boneh in the paper Symmetric Cryptography for JavaScript from 2009 [154]. Although the `SJCL` code base did not see significant changes in the last few years, this library seems to be stable, well-designed, and does not have any known vulnerabilities.

Encryption. We noticed that the web client's source code repository does not include the files `aesasm.js` and `encrypter.js`. We retrieved them using the browser's developer tools when connecting to Mega: the client downloads them as soon as the user uploads a file. The former implements `AES` encryption, including `AES-CCM`, based on a version from 2014 of an open source implementation on GitHub [5]. The latter is responsible for encrypting and uploading chunks of a file. In general, this behavior adds to the

concerns mentioned by Prof. Woodward about in-browser cryptography [95]: a malicious provider could run a targeted attack and replace the encryption code to modify the web client dynamically to disclose information or decryption keys. We selectively verified some of the encryption code and saw that it is the same code that is published on GitHub. Of course, such incidental evidence does not exclude targeted attacks. Mega offers a browser extension that fixes the cryptographic code and only changes it with updates of the extension [62]. Moreover, non-web clients do not load the encryption code dynamically. Nevertheless, we suspect that the majority of the users are agnostic to this issue and use the web client without any additional measures.

The custom AES implementation in `aesasm.js` uses lookup tables. Although very similar code is still present in OpenSSL [36], it was largely superseded by implementations using the native instruction set AES-NI. In addition to the performance gains, another reason for this switch were many side-channel attacks on OpenSSL's AES [94, 74, 86, 75, 112, 80, 91, 162, 149, 111, 83, 115]. In particular, cache observations on the lookup tables, which are also present in `aesasm.js`, leak bits of the key because they are used as indices in the lookup tables. Further work could test whether the classic OpenSSL side-channel attacks can be applied to the in-browser cryptography of Mega. One challenge is to account for the noise introduced by the browser that interprets the JavaScript.

Another observation on the source code is that AES-ECB is used to encrypt the RSA secret key. Since this key is an encoding of the prime factors and secret exponent of a 2048-bit key, the encryption consists of multiple 128-bit AES blocks. The lack of integrity protection in AES-ECB enables tampering with the key, which is only constrained by the key verification function that validates the private key's length encoding. Our attack in Section 4.1 leverages such manipulations to recover the private key. This attack vector has multiple challenges: we do not know the key that the victim recovers from our tampered ciphertext and we can only induce bit flips at AES block granularity in AES-ECB. Furthermore, the usage of RSA encryption is limited to node key sharing, decrypting the session identifier after authentication, and as a fallback for chat key exchange (if no Curve25519 keys are available, the chat keys are encrypted with the recipient's RSA public key).

CSPRNG. While legacy browsers did not provide any secure randomness, modern ones implement `crypto.getRandomValues` to generate cryptographically strong random values [63]. This functionality is present in all major browsers [10].

3.3.8 Privacy

After we have seen Mega's implementation in detail, we will now shift our focus and reflect on Mega's general security properties. We first discuss privacy and then continue with confidentiality (Section 3.3.9) and authentication (Section 3.3.10).

Mega claims to provide privacy by design due to E2EE. While this is true for the uploaded data content, they do admit in section 1.4 of their white paper to store transaction and metadata for "operational and compliance purposes" [62]. According to their GDPR statement, the collected metadata involves access information (browser type, OS, IP address, port), usage statistics (file uploads, folder creations, sharing behavior), file metadata (sizes, timestamps), and communication metadata (time and identity of chat partners, email addresses of contacts) [70]

Despite considering timing side-channels to avoid leaking existing accounts during authentication, there is a direct leak on the recovery page: in order to use the recovery key to reset a password, one needs to enter the account's email address to obtain a reset link. While Mega displays an error message for not registered addresses, they confirm the successful sending of a reset message for registered addresses. One reason why leaking the registered customers can be security critical is targeted spear-phishing: an attacker can validate that a user has a Mega account and pose as a Mega technician or imitate an official maintenance email.

3.3.9 Confidentiality

The overall design of Mega builds on E2EE, which strongly benefits confidentiality: the file data and attributes are encrypted on the client. Therefore, assuming a secure implementation, they are protected in transit and at rest at the cloud provider. In this section, we further comment on the usage of `TLS` and `AES` encryption primitives.

TLS. Mega considers `HTTPS` to only be relevant for loading their home page and API requests. They discourage the use of `HTTPS` for bulk file transfers with the argument that this data is already encrypted. They argue that their other content is not confidential and integrity protected by hashes transmitted over `HTTPS`. We note that integrity is especially relevant for Mega's web client, since it transfers the highly sensitive JavaScript cryptography code over this connection. A successful MitM attacker could modify this code and change the web client to send the master key to the adversary's server. We scanned the supported `TLS` cipher suites of the three primary servers that the Mega web client uses and list them in Appendix C. We note that the more security sensitive domains `g.api.mega.co.nz` and `mega.nz` support

TLS-1.2 and TLS-1.3. The latter includes cipher suites that use AES-CBC and SHA-256. The former may be vulnerable to the plaintext recovering MitM attacks presented by Paterson et. al. in [76]. The *SHAttered* paper [155] found the first collision on SHA-1 in 2017. This hash function is now considered to be deprecated and should be avoided. The server for static content at the domain eu.static.mega.co.nz supports significantly more and older cipher suites. It still offers to run the outdated TLS-1.0 and includes many cipher suites without perfect forward secrecy (i.e., they use static RSA keys for key transfer). Moreover, we observed the command line client to occasionally use HTTP connections for less critical requests unless it is explicitly forced to only use HTTPS.

Encryption Primitives. Mega makes some unusual choices in the cryptographic building blocks: First, they use AES-ECB for encrypting key material and building AES-CTR or CBC-MAC schemes themselves. Second, they use AES-CCM for authenticated encryption. Although there are no known vulnerabilities in AES-CCM, we generally prefer AES-GCM in practice, because it supports buffering data and can be parallelized. We note that Mega's variant of AES-CCM does not implement the standard [164] correctly because they do not encrypt the authentication tag. Therefore, their implementation is an Encrypt-and-MAC scheme instead of MAC-then-Encrypt. However, they later encrypt the tag with a different key as part of the obfuscated file key. This avoids Encrypt-and-MAC's problem of leaking plaintext information via the MAC, such as repeated AES blocks in a file chunk. Nevertheless, the ciphertext is still not integrity protected and the MAC can only be checked after decryption.

Algorithm 11 describes the AES-CCM implementation of Mega's SDK. Consistent with the node encryption in Algorithm 8, they split the iv into the 64-bit file nonce N_F and the 64-bit chunk offset o_i. They compute the CBC-MAC of the message m using $N_F||N_F$ as the IV for the underlying AES-CBC. This deviates from the standard CBC-MAC [103], which does not take an IV as input and instead fixes to the zero byte string. Mega's implementation manually performs AES-CTR encryption of incremented counter blocks and XORs the resulting key stream to the file chunks. Different chunks have distinct counter values for all their AES blocks as long as the entire file has less than 2^{64} AES blocks. Larger files would overflow o_i and lead to repeated key streams, however, this is not a concern for any practical file size. Likewise, the reduction modulo 2^{128} on Line 7 of Algorithm 11 never reduces iv in practice. We remark that Mega deviates from the AES-CCM specification in RFC 3610 [164], by not encrypting the tag τ. According to RFC 3610, Mega would need to use the first key stream block $\texttt{AES-ECB.Enc}(k_F, N_F||0^{64})$ to encrypt the tag and start encrypting the message m with the next key stream block.

Algorithm 11 Mega's custom AES-CCM implementation to encrypt a message m with node key k_F and IV iv

procedure AES_CCM_ENC_MEGA(k_F, m, iv)
  $N_F||o_i \leftarrow iv$ ▷ File nonce N_F, chunk offset o_i
  $\tau \leftarrow$ `CBC-MAC.Tag`$(k_F, m, N_F||N_F)$

  ▷ Mega's `AES-CTR` encryption of m
  $m_0||m_1||\ldots||m_n \leftarrow m$, where $\forall i \in [0,n]\,.|m_i|_2 = 128$
  for all $j \in [0,n]$ **do**
    $c_j \leftarrow$ `AES-ECB.Enc`$(k_F, iv) \oplus m_j$
    $iv \leftarrow \lfloor iv + 1 \rfloor_{2^{128}}$
  return $c_0||c_1||\ldots||c_n, \tau$

3.3.10 Authentication

We discuss three instances relevant for user authentication: the use of the derived authentication key k_a, Two Factor Authentication (2FA), and the account recovery process. Furthermore, we point out the dangerous use of `CBC-MAC` on variable length data.

Authentication Key. Algorithm 7 shows that the client authenticates itself by directly sending k_a to the server. Therefore, a MitM attacker can record k_a and impersonate the user. Since this requires a stronger than usual adversary who compromises the `TLS` connection, we hereafter refer to it as TLS-MitM. Mega argues for this design as a precaution against server database compromise. They only store the truncated `SHA-256` hash, which cannot be used to authenticate, as the user needs to send the plain authentication key. If an attacker still manages to impersonate a user, the server's response is encrypted based on the user password, which is never transmitted. However, knowing k_a allows for a trivial dictionary attack: anyone can request the victim's salt s given its email address. We discussed in Section 3.3.8 how an adversary could detect registered email addresses. Using s and a list of frequently used passwords (for instance, the RockYou password list), the adversary can run `PBKDF2` with 100'000 iterations. The attack has identified the correct password when the least significant 256 bits of the derived 512-bit key correspond to k_a. Since `PBKDF2` has moderate memory requirements, GPUs and customized hardware can be used to parallelize the key derivation and crack the password in reasonable time [100]. This attack cannot only be performed by Mega themselves but also by any entity compromising Mega's server and observing a connection request, by a TLS-MitM adversary, or by any attacker tricking the victim into disclosing k_a (e.g., by redirecting a client to a malicious website with DNS poisoning).

2FA. Although Mega supports 2FA, this can be deactivated by their staff. Therefore, an additional factor only increases our security against outsider attacks and not against a malicious provider.

Account Recovery. Since the authentication key k_a is derived from the password, account recovery has to use an alternative way to authenticate users. Resetting the password or deactivating the account both work by sending an email to the user, which provides a link to perform this action. As described in Section 3.3.5, the password reset requires the recovery key, which is the user's master key. Therefore, despite being able to intercept authentication emails, Mega can only deactivate an account, but not set a new password. However, they could replace the account with a new one that uses the same email address. This would trigger a warning for chat partners that the public key's fingerprint changed, and remove established contact relationships.

3.3.11 Integrity.

We make the following remarks about the integrity protection provided by Mega.

CBC-MAC. Algorithm 11 shows that Mega uses `CBC-MAC` to compute an authentication tag over the variable-length concatenation of file chunk MACs to create the condensed MAC M_{cond}. NIST recommends `CMAC` for authentication in NIST SP 800-38B [102] because textbook `CBC-MAC` is vulnerable to truncation attacks. However, there is no immediate attack because the individual chunk MACs are not uploaded to the server. Moreover, the same issue arises for computing the chunk MACs themselves, however, the web client and SDK use the same chunk size for the entire file.

File Attributes. File attributes are encrypted with `AES-CBC` and therefore not integrity protected. Mega could try to tamper with the file name or serve different attributes, since they are not associated to the file content. However, the IV is fixed and therefore, Mega cannot leverage the malleability of `AES-CBC` to make controlled changes.

Chapter 4

Attacks on Mega

We present multiple attacks in a threat model with a malicious cloud provider or a TLS-MitM adversary (i.e., a strong MitM attacker who compromises the TLS connection). Section 4.1 shows a practical RSA key recovery attack combining key overwriting capabilities and properties of the RSA-CRT decryption to perform a binary search for one of the factors of the RSA modulus. The recovered RSA private key enables the adversary to decrypt key material for nodes that were shared with the victim. Building on this key recovery attack, Section 4.2 presents a plaintext recovery attack, which decrypts two blocks encrypted with AES-ECB under the master key in a single query. This attack enables the malicious cloud provider to recover the victim's file encryption keys, chat key, and signature key. Building on this decryption attack, Section 4.3 shows how a malicious cloud provider can stealthily plant a new file in the victim's cloud storage. Furthermore, Section 4.4 presents a more expensive RSA decryption attack that uses a different attack vector than Section 4.1 and contributes a novel variation of Bleichenbacher's attack on PKCS#1 v1.5 [88] adapted to Mega's custom padding and taking into account an unknown prefix.

Throughout this chapter, we assume the standard notation for RSA with the modulus $N = p \cdot q$ for two large primes p and q. Furthermore, we have the public exponent e and the secret exponent $d = e^{-1} \mod \varphi(N)$ for $\varphi(N) = (p-1) \cdot (q-1)$.

4.1 Mega RSA Key Recovery Attack

We present a practical attack to recover a user's private key by factoring the RSA modulus. A malicious provider or a TLS-MitM adversary can use the RSA private key to decrypt node sharing key material. If the legacy RSA key transfer was used to exchange chat keys with the victim, the adversary can additionally recover these chat keys. Furthermore, a TLS-MitM attacker

can decrypt future SIDs, which enables passive eavesdropping on the session. Section 4.1.1 starts by describing the RSA encryption and padding of Mega. After introducing the setting in Section 4.1.2, we provide the attack in Section 4.1.3 and an optimization to increase its performance and reduce its detectability in practice in Section 4.1.4. Finally, Section 4.1.5 explains our proof of concept setup. Furthermore, Section 4.2 describes an attack to decrypt arbitrary AES-ECB ciphertext blocks encrypted under the master key, which builds on the RSA key recovery attack. Section 4.3 combines these two attacks to frame users with maliciously inserted files.

4.1.1 Mega's RSA Encryption

We start by presenting Mega's usage of RSA encryption. We recall from Algorithm 4 that the user's client generates a 2048-bit RSA key pair (sk_{share}, pk_{share}) during registration. The client stores the secret key sk_{share} on Mega's servers after encrypting it with AES-ECB under the master key k_M. The server sends the encrypted private key to the client to support multiple devices. The client uses RSA to receive key material shared by a contact and as a fallback for exchanging symmetric chat keys. Our attack targets the RSA decryption of the session ID that the server sends after authentication.

Algorithm 12 shows our reconstruction of Mega's RSA encryption and decryption of the session ID *sid* to the best of our knowledge. The padding values (r_1 and r_2) and the generation of the 43-byte *sid* are left undefined because Mega does not publish their server code. However, the need for compatibility with the client-side decryption determines the position of the SID. Moreover, the client does not check the padding and our attack replaces *sid* independent of how the original value was chosen. Therefore, any instantiation of this algorithm works for our attack.

Algorithm 12 Mega's RSA encryption of the SID

1: **procedure** MEGA_RSA_ENCRYPT_SID(pk_{share})
2: $sid \in [0,255]^{43}$ ▷ E.g., $sid \leftarrow\!\!\$\ [0,255]^{43}$
3: $sid_{padded} \leftarrow r_1||sid||r_2$ ▷ For some $r_1 \in [0,255]^2, r_2 \in [0,255]^{211}$
4: $N, e \leftarrow pk_{share}$
5: **return** $\left(sid_{padded}\right)^e \mod N$

The client encrypts the following encoding of the RSA private key with AES-ECB under the master key k_M:

$$sk_{share}^{encoded} \leftarrow l(q)||q||l(p)||p||l(d)||d||l(u)||u$$

where $l(x)$ is the two-byte big-endian length encoding of the byte-length of x and $u \leftarrow \lfloor q^{-1} \rfloor_p$. The length encoding is used to split the secret key

components during decoding. The client pads $sk_{share}^{encoded}$ to a multiple of the AES block size. For 2048-bit RSA, $sk_{share}^{encoded}$ contains four length encodings of 2 B each, three (close to) 128 B elements (q, p, and u), and the private exponent of approx. 256 B. In the previous size estimations, we use the rough approximation that inverses modulo a random x-bit number are close to uniformly distributed and, thus, have close to x bits with high probability[1]. Since AES blocks are 16 B (128 bits), this results in 41 blocks, where the last one has eight bytes of padding.

Algorithm 14 shows the RSA decryption of the SID sid. The server sends the encrypted session ID c_{sid} and RSA private key $c_{sk_{share}^{encoded}}$. The client first attempts to decrypt the latter using its master key k_M. Since there is no integrity protection, it can only decode the AES-ECB-decryption result, and sanity check the length encodings as shown in Algorithm 13. The inverse $l^{-1}(b)$ of the length encoding function l converts two bytes b from big-endian encoding to an integer. We remark that the client neither verifies the padding nor the lengths of the individual components. Instead, it ensures that the private key contains four length encodings and components, and parses them. After decoding the private key, the client performs RSA-CRT decryption, which saves a factor of four compared to the computation time of naïve decryption $\lfloor (c_{sid})^d \rfloor_N$. The additional values d_p and d_q for RSA-CRT are precomputed once per session at the end of Algorithm 13 and reused for every subsequent decryption. Lines 7 and 8 of Algorithm 14 decrypt the padded session ID m in the smaller rings $\mathbb{Z}_p$ and $\mathbb{Z}_q$. Note that the exponents d_p respectively d_q have about half the bit size of d. Next, Lines 9 to 11 recover the padded session ID m modulo N using Garner's formula [107] for the CRT. Instead of checking the padding, the code uses the known session ID length to truncate the decryption m to sid'. Before the truncation, m is left padded with zero bytes until it has length $|N|_8$. Next, the unpad operation removes the trailing $|N|_8 - 45$ bytes of padding as well as the leading two bytes. Line 12 specifies this extraction of the 43-byte sid' by operating on integers instead of bytes. In a correct execution, we have $m = sid_{padded}$ and $sid' = sid$. The client sends sid' in subsequent requests to the server to authenticate itself.

4.1.2 Threat Model

In our attack, we assume a malicious service provider. A slight variation considers a TLS-MitM adversary. Note that the server controls the SID and its padding. Moreover, the attacker has access to a partial decryption oracle since the client returns the truncated session ID if its decryption was successful. Finally, the adversary can garble AES blocks of the private key due to the lack of integrity protection in AES-ECB.

[1] As a rough approximation, the average bit size of a random number in $[0, 2^x)$ is $2^{-x}\left(2 + \sum_{i=2}^{x} i \cdot 2^{(i-1)}\right) = 2^{-x} \cdot (2 + x2^x - 2^x) \approx x - 1$.

Algorithm 13 Parses the encoded RSA private key into the components q, p, d, and u, and constructs the missing components N, e, d_p, and d_q for RSA-CRT.

1: **procedure** DECODE_RSA_PRIVK($sk_{share}^{encoded}$)
2: $L \leftarrow 0$
3: **for** $i \in [0, 3]$ **do**
▷ Parse length encoding
4: **if** $L + 2 > |sk_{share}^{encoded}|_8$ **then return** $\perp$
5: $l_i \leftarrow l^{-1}(sk_{share}^{encoded}[L{:}L+2])$
6: $L \leftarrow L + 2$

▷ Parse private key component
7: **if** $L + l_i > |sk_{share}^{encoded}|_8$ **then return** $\perp$
8: $x_i \leftarrow sk_{share}^{encoded}[L{:}L+l_i]$
9: $L \leftarrow L + l_i$

▷ Allow at most one AES block of padding
10: **if** $|sk_{share}^{encoded}|_8 - L > 16$ **then return** $\perp$

▷ Derive RSA-CRT components
11: $q, p, d, u \leftarrow x_0, x_1, x_2, x_3$
12: $N \leftarrow p \cdot q$
13: $e \leftarrow \lfloor d^{-1} \rfloor_{\varphi(N)}$
14: $d_p \leftarrow \lfloor d \rfloor_{p-1}$
15: $d_q \leftarrow \lfloor d \rfloor_{q-1}$
16: **return** $N, e, d, p, q, d_p, d_q, u$

Section 4.1.3 explains how we leverage the above capabilities to factor the RSA modulus. Section 4.1.4 reduces the number of client logins required for the attack.

4.1.3 Factoring the RSA Modulus Using Binary Search and Robustness Properties of RSA-CRT

We start by investigating the effect on the RSA-CRT decryption when we garble the value u from the private key. We can change u at the granularity of AES blocks: e.g., modifying the second to last ciphertext block[2] of $c_{sk_{share}^{encoded}}$ changes the 128 bits of the corresponding plaintext block. Since the AES block cipher is a pseudo-random permutation, we cannot predict the resulting block. However, it suffices for our attack that the client recovers $u' \neq \lfloor q^{-1} \rfloor_p$.

[2]This is a better choice than garbling the last block, because it does not change the padding. Thus, the attack still works, even if clients verify the padding in the future.

Note that any non-identity transform of the ciphertext block is guaranteed to fulfill this property, since AES is a permutation and, therefore, two different ciphertext blocks are guaranteed to decrypt to different plaintext blocks.

Algorithm 14 Mega's RSA decryption of the encrypted SID c_{sid} using the user's encrypted private key $c_{sk_{share}^{encoded}}$

1: **procedure** MEGA_RSA_DECRYPT_SID($c_{sk_{share}^{encoded}}$, c_{sid})
2: $sk_{share}^{encoded} \leftarrow$ AES-ECB.Dec($k_M, c_{sk_{share}^{encoded}}$)
3: $\gamma \leftarrow$ decode_rsa_privk($sk_{share}^{encoded}$)
4: **if** $\gamma = \perp$ **then**
5: **return** $\perp$ ▷ Private key parsing failed
6: $N, e, d, p, q, d_p, d_q, u \leftarrow \gamma$

▷ RSA-CRT decryption
7: $m_p \leftarrow \left\lfloor (c_{sid})^{d_p} \right\rfloor_p$
8: $m_q \leftarrow \left\lfloor (c_{sid})^{d_q} \right\rfloor_q$
9: $t \leftarrow \left\lfloor m_p - m_q \right\rfloor_p$
10: $h \leftarrow \left\lfloor t \cdot u \right\rfloor_p$
11: $m \leftarrow h \cdot q + m_q$ ▷ Equal to $\left\lfloor (c_{sid})^{d} \right\rfloor_N$
12: $sid' \leftarrow \left\lfloor \left\lfloor \frac{m}{256^{|N|_8 - 45}} \right\rfloor \right\rfloor_{256^{43}}$ ▷ Simplification for $|m|_2 < 256^{254}$
13: **return** sid'

We distinguish two cases for the decryption: $m < q$ and $m \geq q$. Note that RSA-CRT is not symmetric. Garner's formula selects one prime q for which it directly adds m_q to the result and then computes the multiple h of q required to satisfy the second equation for m_p. Our attack always recovers q, independent of whether it is the larger or smaller prime factor of N.

Case $m < q$. First, we note that $m_q = m$ because by the CRT $m \equiv_q m_q$ and since $m < q$ we have $\lfloor m \rfloor_q = m$. As $m_q < q$ by definition, we conclude that $m_q = m$. For m_p, we again have by the CRT that $m \equiv_p m_p$. Therefore, there exists $\alpha \in \mathbb{Z}$ such that $m = m_p + \alpha \cdot p$. Combining these observations on m_q and m_p, we have for t on Line 9 of Algorithm 14 that $m_p - m_q = m - \alpha \cdot p - m = -\alpha \cdot p$ and thus $t = \lfloor m_p - m_q \rfloor_p = 0$. Therefore, $h = 0$, independent of the value of u. In other words, the client recovers the correct result $m' = h \cdot q + m_q = m_q = m$ despite the modified u value.

Case $m \geq q$. It directly follows that we have $m_q \neq m$ from $m_q < q \leq m$. There exist $\alpha, \beta \in \mathbb{Z}$ such that $m_p = m - \alpha \cdot p$ and $m_q = m - \beta \cdot q$.

Therefore, $t = \lfloor -\alpha \cdot p + \beta \cdot q \rfloor_p = \lfloor \beta \cdot q \rfloor_p$. Since p and q are coprime, $t \neq 0$ iff $\gcd(\beta, p) = 1$. This happens with overwhelming probability $1 - 1/p$. Therefore, with high probability, $h \neq 0$ and $m' = h \cdot q + m_q \neq m$. Although $m' \equiv_q m_q$, we have $m' \not\equiv_p m_p$ because $u \neq \lfloor q^{-1} \rfloor_p$ and, therefore, Lines 9 to 11 no longer compute the CRT.

We observe that if the adversary would be able to obtain the full value m', then it could easily distinguish the two cases (w.h.p.): if $m' = m$, then we have $m < q$, otherwise $m \geq q$. This oracle on the size of q allows the attacker to perform a binary search for the factor q. It can halve the interval containing q by querying the middle value as the padded session ID m. Although the benign encoding of *sid* right pads it so that m is close to the modulus, there is no padding check on the client. Due to this implementation, any integer $m \in [0, N)$ is a valid message.

However, the adversary only sees the unpadded sid' instead of m'. We notice that for $m < q$ we have $sid' = 0$, because $m' < 256^{|N|_8 - 45}$ and we divide m' by $256^{|N|_8 - 45}$ on Line 12. This upper bound on m' holds because $q < 2^{|N|_2/2}$ since Mega's RSA key generation algorithm selects primes of exactly $|N|_2/2$ bits (where $|N|_2$ is a power of two). For 2048-bit RSA, we clearly have $m' < q < 2^{|N|_2/2} = 2^{1024} < 2^{1688} = 256^{|N|_8 - 45}$. Moreover, we observe that $m' \geq 256^{|N|_8 - 45}$ with high probability if $m \geq q$. The reason is that m' has the summand $h \cdot q$, where we argued above that $h \neq 0$ with probability $1 - 1/p$. In fact, h is a random number of approx. $|N|_2/2$ bits[3]. Since q has $|N|_2/2$ bits, $h \cdot q$ has approx. $|N|_2$ bits and, thus, m' is w.h.p. larger than $256^{|N|_8 - 45}$. More precisely, the probability of a false positive is:

$$\Pr\left[h \cdot q < 256^{|N|_8 - 45}\right] \leq \Pr\left[|h \cdot q|_8 < |N|_8 - 45\right] \leq \Pr\left[|h|_8 < \frac{|N|_8}{2} - 45\right]$$

For RSA-2048, this is $\Pr[|h|_8 < 83] \approx 256^{83} \cdot 256^{-128} = 2^{-360}$, if we assume $h \leftarrow\!\!\$\ \{0,1\}^{1024}$ (which ignores the slight bias of small numbers due to the wrap around modulo p).

In conclusion, despite the truncated decryption oracle, our adversary can distinguish the two cases with overwhelming probability.

Algorithm 16 summarizes the binary search attack on a higher level by using the comparison oracle $\mathcal{O}_{comp}$ specified in Algorithm 15. $\mathcal{O}_{comp}$ returns true with high probability if $m < q$. In our model, the adversary can implement this oracle because he has access to the encrypted RSA secret key $c_{sk_{share}^{encoded}}$: Mega stores this value and transmits it on every authentication. Algorithm 15

[3] h is a close to uniformly random number in $[0, p]$ (there is a very slight bias towards smaller numbers due to a possible wrap around). As a rough approximation, the average bit size of a random number in $[0, 2^x)$ is $2^{-x}\left(2 + \sum_{i=2}^{x} i \cdot 2^{(i-1)}\right) = 2^{-x} \cdot (2 + x2^x - 2^x) \approx x - 1$. Similarly, we expect h to have close to $|N|_2/2$ bits w.h.p.

Algorithm 15 Abstract oracle comparing a query m with the prime factor q of N and returning true if $m < q$ w.h.p.

procedure $\mathcal{O}_{comp}$(m)
  Known: pk_{share}, $c_{sk_{share}^{encoded}}$
  $c'_{sk_{share}^{encoded}} \leftarrow c_{sk_{share}^{encoded}} \oplus r \cdot 2^{128}$ ▷ For some $r \in (\{0,1\}^{128} \setminus \{0\})$
  $c_{sid} \leftarrow \texttt{RSA.Enc}(pk_{share}, m)$
  $sid' \leftarrow \texttt{mega\_rsa\_decrypt\_sid}(c'_{sk_{share}^{encoded}}, c_{sid})$
  return $sid' = 0$

first modifies the second to last ciphertext block to garble u used in the RSA-CRT decryption. Next, it sends the RSA encryption of m as the encrypted session ID and lets the victim execute Algorithm 14 on the prepared inputs. The victim responds by sending sid', where $sid' = 0$ means that $m < q$ with high probability. For the sake of presentation simplicity, we assume in Algorithm 16 that $|N|_2$ is a power of two and that the primes p and q have exactly $|N|_2/2$ bits (which is the case for Mega). The initial interval contains all $|N|_2/2$-bit values. Next, the attack continues to query the middle value of the current interval and updates the lower bound a or the upper bound b of the interval depending on the query result. We stop when only one value – the solution q – remains in the interval.

Algorithm 16 Binary search key recovery attack on Mega's RSA encryption/decryption of the SID using the comparison oracle $\mathcal{O}_{comp}$.

procedure MEGA_KEY_RECOVERY_ATTACK($\mathcal{O}_{comp}$)
  $a, b \leftarrow 2^{\frac{|N|_2}{2}-1}, 2^{\frac{|N|_2}{2}}$ ▷ Invariant: $q \in [a, b)$
  while $a \neq b$ **do**
    $m \leftarrow \left\lfloor \frac{a+b}{2} \right\rfloor$
    if $\mathcal{O}_{comp}(m)$ **then** ▷ $\mathcal{O}_{comp}(m)$: **return** $m < q$
      $a \leftarrow m$
    else
      $b \leftarrow m$
  return a

4.1.4 Reducing the Number of Queries

The attack presented in Section 4.1.3 requires $\frac{|N|_2}{2} - 1$ queries due to the binary search on an interval of the size $2^{\frac{|N|_2}{2}} - 2^{\frac{|N|_2}{2}-1} = 2^{\frac{|N|_2}{2}-1}$. For 2048-bit RSA, this involves 1023 queries. If the adversary is the service provider, it can stealthily mount the attack by accepting any session ID returned by the victim. However, a TLS-MitM adversary does not know the correct SID,

because it cannot decrypt c_{sid} itself, and it already convinced the victim to decrypt and store a different value. Consequently, the user needs to perform an additional login after every attacker query. This section aims to reduce the number of required login requests to make the attack faster in practice and less noticeable in the TLS-MitM scenario.

For this purpose, we terminate the binary search as soon as the interval is "small enough" and use lattices to factor the modulus given the leading bits of one prime factor. In the following, we describe the straightforward application of a lattice with small dimension adapted from the explanation in section 4.2.2 of Gabrielle et al. [98]. This approach works for up to $\lfloor log_2(N^{1/6}) \rfloor$ unknown bits. Therefore, we reduce the required number of queries to $\frac{|N|_2}{2} - \left\lfloor \frac{|N|_2 - 1}{6} \right\rfloor$, which corresponds to 683 queries in the case of `RSA`-2048. With more complex lattices described by Howgrave-Graham [113] and May [138], it is possible to recover up to $\lfloor log_2(N^{1/4}) \rfloor$ unknown bits (512 queries for `RSA`-2048).

Lattice Attack. Assume we know the leading bits $\hat{p}_1$ of the prime $p \leftarrow \hat{p}_1 || p_0$, where $l \leftarrow |p_0|_2$ bits are unknown. Let $p_1 \leftarrow \hat{p}_1 \cdot 2^l$ be the integer shifted to the left by l bits such that $p = p_1 + p_0$. On a high level, we rewrite the problem of recovering p_0 to finding small roots of a polynomial. For this purpose, we consider the following three polynomials f_1, f_2, and f_3 over $\mathbb{Z}$, which all have the small root p_0 modulo p:

$$
\begin{aligned}
f_0(x) &= x \cdot (p_1 + x), & f_1(p_0) &= p_0 \cdot (p_1 + p_0) = p_0 \cdot p \equiv_p 0 \\
f_1(x) &= p_1 + x, & f_0(p_0) &= p_1 + p_0 = p \equiv_p 0 \\
f_2(x) &= N, & f_2(p_0) &= N = p \cdot q \equiv_p 0
\end{aligned}
$$

We observe that every linear combination of these polynomials has the same root p_0 modulo p. Another way to express this is using coefficient vectors $[v_2, v_1, v_0] \in \mathbb{Z}^3$ to represent any polynomial of degree two: $v_2 \cdot x^2 + v_1 \cdot x + v_0$. We can easily see that the addition of polynomials corresponds to coefficient vector addition. The polynomials f_2, f_1, and f_0 correspond to the coefficient vectors $[1, p_1, 0]$, $[0, 1, p_1]$, and $[0, 0, N]$. The basis consisting of these three coefficient vectors spans the vector space containing all linear combinations of f_2, f_1, and f_0. We use this observation to construct the following lattice basis B, where we put the coefficient vectors of our polynomials in the rows and scale the first column by L^2 and the second by L, where $L = 2^l$:

$$
B = \begin{bmatrix} L^2 & Lp_1 & 0 \\ 0 & L & p_1 \\ 0 & 0 & N \end{bmatrix}
$$

The columns are scaled in this way to make the `L1` norm of any vector in the lattice an upper bound on the value of the corresponding unscaled

polynomial evaluated at p_0. In more detail, for the coefficients $\alpha_2, \alpha_1, \alpha_0 \in \mathbb{Z}$, we have the following vector w in the lattice:

$$w = [\alpha_2, \alpha_1, \alpha_0] \cdot B = [\alpha_2 \cdot L^2, (\alpha_2 \cdot p_1 + \alpha_1) \cdot L, \alpha_1 \cdot p_1 + \alpha_0 \cdot N]$$

The corresponding unscaled polynomial g is:

$$g(x) = v_2 \cdot x^2 + v_1 \cdot x + v_0, \text{ for } \begin{cases} v_0 = \alpha_1 \cdot p_1 + \alpha_0 \cdot N \\ v_1 = \alpha_2 \cdot p_1 + \alpha_1 \\ v_2 = \alpha_2 \end{cases}$$

We have $g(p_0) = v_2 \cdot (p_0)^2 + v_1 \cdot p_0 + v_0 < v_2 \cdot L^2 + v_1 \cdot L + v_0 = ||w||_1$, since $p_0 < L = 2^l$ because $|p_0|_2 = l$.

Consequently, if we can find a vector w with $||w||_1 < p$ in the lattice, then there is a corresponding unscaled polynomial g, such that $g(p_0) < p$. Furthermore, p_0 is a root of g modulo p by construction because g is a linear combination of f_2, f_1, and f_0. Thus, from $g(p_0) < p$ and $g(p_0) \equiv_p 0$, it follows that $g(p_0) = 0$ over the integers. Polynomials can be factored in polynomial-time [136], which enables us to efficiently compute the roots of g and test which root r satisfies $1 < \gcd(p_1 + r, N) < N$. That r consists of the missing bits p_0 of p and $p = p_1 + r$.

It remains to show that our lattice contains a sufficiently small vector w. Minkowski's first theorem states that the shortest vector in the vector space spanned by basis B has the length $||w||_2 \leq \sqrt{n} \cdot \det(B)^{1/n}$, where n is the dimension of B [142]. Although the shortest vector problem is NP-hard for arbitrary lattices, the LLL algorithm [136] by Lenstra, Lenstra, and Lovász finds an exponential approximation of the shortest vector in polynomial time [98]. Nguyen and Stehlé showed that the vector w found by LLL satisfies $||w||_2 \leq 1.02^n \det(B)^{1/n}$ on average for random lattices. Our lattice basis B is of dimension $n = 3$ and has determinate $\det(B) = L^3 N$. Using $||w||_1 \leq \sqrt{n} \cdot ||w||_2$, we therefore derive $||w||_1 \leq \sqrt{3} \cdot 1.02^3 \cdot L \cdot N^{1/3}$. In order to find $||w||_1 < p$, we arrive at the following condition, using $p < 2\sqrt{N}$ as bound[4]:

$$\sqrt{3} \cdot 1.02^3 \cdot L \cdot N^{1/3} < p < 2\sqrt{N} \Longrightarrow L < \frac{2}{\sqrt{3} \cdot 1.02^3} N^{1/6} \Longrightarrow l \approx |N^{1/6}|_2$$

For 2048-bit `RSA`, we can therefore recover approximately $l = 341$ bits. Hence, we can stop the binary search when $b - a < 2^l$ for the interval $[a, b)$. In that case, either the upper 683 bits of a or of b are the most significant bits of p. To explain this, we first observe that any number of upper bits of a and b

[4]For Mega, the prime factors of N have exactly $|N|_2/2$ bits and $|N|_2$ is a power of two.

can be different despite these numbers being close together. For instance, all leading 683 bits of $a \leftarrow 2^{1023} - 1$ and $b \leftarrow 2^{1023}$ differ despite $b - a = 1$. However, there cannot be more than one change to the bits above the least significant l bits, because otherwise the difference between a and b would be larger than 2^l. Therefore, all values in $[a, b)$ either start with the leading bits of a or b. We run the lattice attack for both prefixes and identify the correct prime p by verifying that it factors the modulus N. We empirically verified that we can successfully factor N for $l = |N^{1/6}|_2$ unknown bits in 1000 out of 1000 runs with 683 queries.

In conclusion, this optimization reduces the number of queries for our attack from 1023 to 683. As mentioned previously, higher-dimensional lattices [113, 138] would allow us to further decrease the number of queries to 512 at the limit.

4.1.5 Proof of Concept Attack

This section describes our proof of concept implementation of the attack. Since Mega does not publish their server's source code, we realize the attack with a TLS-MitM attacker who hijacks the victim's connection to Mega's servers and patches their responses on the fly.

Figure 4.1 shows the (simplified) requests of a normal successful login attempt. The client starts by making a connection request to `mega.nz`. In response, it receives the static content of the landing page. The client initiates the login procedure by sending the user's email address to receive the salt. The client derives the decryption and authentication keys from the salt and the user's password and sends the authentication key to the server (see Algorithm 5 for a detailed description of this process). After successfully verifying the client's identity, the server sends the client's encrypted private RSA key $c_{sk_{share}^{encoded}}$ and c_{sid}, the session ID encrypted for this RSA key. The client uses Algorithm 14 to decrypt the SID. The client includes the session ID to authenticate itself to the server in subsequent requests. In this session, the client can obtain more account information or request the user's encrypted files.

Figure 4.2 visualizes our TLS-MitM key recovery attack realizing the procedure from Section 4.1.4 in practice. We use `mitmproxy`[5], an interactive HTTPS proxy, to implement the adversary. After installing a trusted root certificate in the victim's root of trust, `mitmproxy` forges TLS certificates on the fly for all requests routed over it. It is rather sophisticated and pauses the connection, requests the benign certificate, and creates a forgery with the correct common name, subject alternative names, and other properties set as in the genuine certificate. Moreover, it exposes an API and allows

[5] https://mitmproxy.org/

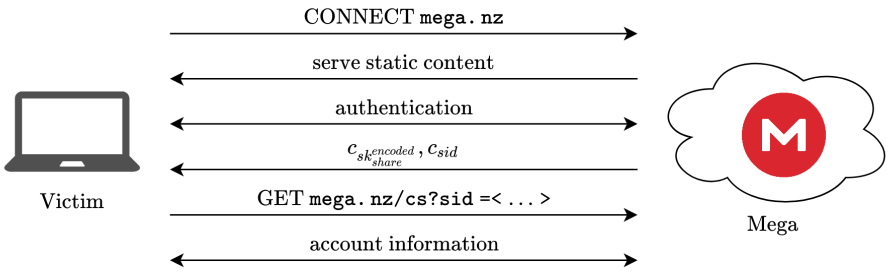

Figure 4.1: Visualization of a normal authentication and login between a client and Mega's servers.

registering Python scripts. Therefore, we implement our attack in Python and Sage and register listeners for the server response containing $c_{sk_{share}^{encoded}}$ and c_{sid}. We intercept these values, garble the second to last AES block in $c_{sk_{share}^{encoded}}$ and send c_m, the center m of the current interval encrypted under the victim's public key, to the client (cf. Section 4.1.3). Next, we intercept the first request where the victim includes the SID. If it is the URL Base64 encoding of zero, we know $m < q$ and otherwise $m \geq q$.

To test the entire attack end-to-end, we create a test account on Mega. We implement a victim who uses Selenium[6] to automate a Firefox browser and request Mega's login page, enter the test account's credentials, and log in. We route the victim's requests over `mitmproxy` and observe that our binary search for a factor of the RSA modulus is successful. Furthermore, we empirically verify the lattice attack from Section 4.1.4 in an offline simulation to successfully recover 341 bits of a factor of the modulus for RSA-2048 in 1000 out of 1000 runs. Together with 683 queries from the binary search attack, we can therefore factor the RSA-2048 modulus and recover the private key.

We remark that we could improve this MitM attack to not be detectable from the server. The adversary can store the correct c_{sid} value sent by the server and subsequently impersonate the server for all requests during the binary search attack. When the adversary decides to allow the victim to log in successfully, it can send the original c_{sid} and stop intercepting traffic. For the server, this appears as if a slightly slower client made a single login attempt. The MitM can choose a tradeoff between attack speed and detectability: more queries lead to a quicker factorization of the RSA modulus. However, they cause more failed login attempts at the client and a more noticeable authentication delay for the server.

[6] https://www.selenium.dev/

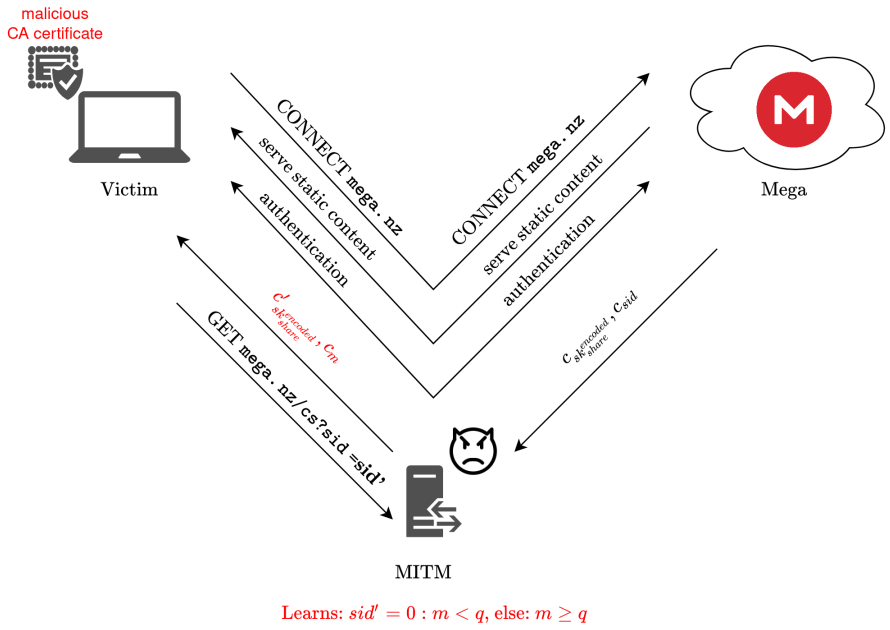

Figure 4.2: Visualization of the key recovery proof of concept attack by a TLS-MitM adversary.

4.2 Mega Plaintext Recovery for AES-ECB Under the Master Key

We build on the RSA private key recovery attack from Section 4.1 and use the recovered RSA private key, together with a similar attack vector consisting of key overwriting and RSA-CRT, to decrypt AES-ECB ciphertexts. Since Mega protects all key material with AES-ECB under the master key, we can decrypt node keys, the victim's signing key, and its chat keys. A single hijacked login attempt suffices to decrypt two AES blocks, which corresponds to a full node key. Section 4.2.1 recalls the setting and introduces some notation, and Section 4.2.2 describes the plaintext recovery attack.

4.2.1 Threat Model and Preliminaries

We consider a malicious cloud service provider who recovered the victim's private RSA key with the attack discussed in Section 4.1 and who has access to the victim's encrypted key material. Clients store the latter on Mega's server to support access from multiple devices[7]. The adversary aims to recover the plaintexts $pt_i \leftarrow \texttt{AES-ECB.Dec}(k_M, ct_i)$ for $i \in \{0,1\}$, which are the decryption

[7]The case study in Section 3.3 provides a detailed description of Mega's design.

of the the target ciphertexts ct_i encrypted with AES-ECB under the master key k_M.

We recall that the victim encodes its RSA private key as follows before encrypting it with AES-ECB under the master key k_M:

$$sk_{share}^{encoded} \leftarrow l(q)||q||l(p)||p||l(d)||d||l(u)||u$$

where $l(x)$ is the two-byte big-endian length encoding of the byte length of x and $u \leftarrow \lfloor q^{-1} \rfloor_p$. We split this encoding into 128-bit AES blocks for the AES-ECB encryption:

$$b_0||b_1||\ldots||b_\alpha \leftarrow sk_{share}^{encoded}, \text{ where } \forall i \in [0,\alpha]\,.\,|b_i|_2 = 128$$
$$c_i \leftarrow \texttt{AES-ECB.Enc}(k_M, b_i) \forall i \in [0,\alpha]$$

For instance, we have $b_0 \leftarrow l(q)||q[0{:}13]$ and $b_1 \leftarrow q[14{:}29]$. We focus on the case where d requires 256 bytes and u needs 128 bytes to simplify the analysis. Consequently, the RSA-2048 private key encoding consists of $\alpha = 41$ AES blocks with eight bytes of padding in the last block b_{40}. We are particularly interested in the first block containing bits of u for our attack: $b_{32} \leftarrow d[250{:}255]||l(u)||u[0{:}7]$. Although deviations in the size of d and u are possible[8], they are very unlikely to change $|d|_8$ or $|u|_8$ because inverses are roughly uniformly distributed and thus close in bit size to their modulus. Nevertheless, it is straightforward to adapt our attack to corner cases with shorter encodings of d or u.

4.2.2 Attack Description

We focus in this section on attacking RSA-2048, which Mega uses. Straightforward modifications of the attack suffice to adapt it for different key sizes.

Key Overwriting. In the first step of our attack, we overwrite ciphertext blocks c_{33} and c_{34} with our target blocks ct_0 and ct_1. Let $u_0 \leftarrow u[0{:}7]$, $u_1 \leftarrow u[8{:}39]$, and $u_2 \leftarrow u[40{:}127]$. Then, we have $u = u_0||u_1||u_2$, $b_{32} = d[250{:}255]||l(u)||u_0$, and $b_{33}||b_{34} = u_1$. By replacing the ciphertext blocks c_{33} and c_{34}, the AES-ECB decryption recovers some $u' \leftarrow u_0||x||u_2$. Our goal is to recover x, which corresponds to $pt_0||pt_1$. We leverage that the RSA-CRT decryption of a ciphertext of our choosing uses u'. Since we have already recovered the victim's private key, we know u_0 and u_2 together with the RSA private key. The client sends a truncation of the RSA-CRT decryption as session identifier to the server. The adversary can observe this SID and, as explained in the following paragraphs, recover x.

[8] A value $y \leftarrow x^{-1} \mod n$ for some $n \in \mathbb{Z}$ and $x \in \mathbb{Z}_n$ has $|n|_8 - 1$ bytes with probability approximately 2^{-8}, because half of the values have the first bit set to one assuming roughly uniform distribution.

Simplifying RSA-CRT. We choose $m \leftarrow u \cdot q$ as the "session ID" that we encrypt using the victim's RSA public key pk to $c_m \leftarrow \texttt{RSA.Enc}(pk, m)$. We send c_m together with the tampered private key encryption from the previous paragraph to the victim. The RSA-CRT decryption of Mega's clients in Algorithm 14 derives on Line 7 and Line 8 that $m_p = \lfloor (c_m)^d \rfloor_p = \lfloor u \cdot q \rfloor_p = 1$ (because $u = \lfloor q^{-1} \rfloor_p$) and $m_q = \lfloor (c_m)^d \rfloor_q = \lfloor u \cdot q \rfloor_q = 0$. These steps are not affected by the modified private key, since they only use the preserved values p, q, and d from the RSA private key encoding. Furthermore, $m_p = 1$ and $m_q = 0$ lead to $t = 1$ and $h = \lfloor u' \rfloor_p$. Consequently, the decryption result for c_m simplifies to $m' = \lfloor u' \rfloor_p \cdot q$ for u' prepared as discussed above with the two target ciphertexts. We now argue that with high probability, this corresponds to $m' = u' \cdot q$. Intuitively, this is desirable for the adversary because it can recover $u' \leftarrow m'/q$. We discuss below how we can recover x from a truncated m'. To argue that $u' < p$ w.h.p., let $p_0 || p_1 \leftarrow p$, where $|p_0|_2 = |u_0|_2$. Since u and u' both start with u_0 and $u < p$, we can only have $u' \geq p$ if $u_0 = p_0$. Thus, we derive the following lower bound for $\Pr[u' < p]$:

$$\Pr[u' < p] \geq 1 - \Pr[u_0 = p_0] = 1 - \frac{p_1 + 1}{p} \geq 1 - 2^{-63}$$

Where we use in the last step that the 1024-bit prime p satisfies $p > 2^{1023}$ and $p_1 + 1 \leq 2^{960}$, because p_0 is eight bytes large. Together, we conclude that $\frac{p_1+1}{p} \leq 2^{960} \cdot 2^{-1023} = 2^{-63}$.

Recovering x. Instead of m', the victim only returns the 43-byte session ID to the server. With probability $1 - 2^{-8}$, the truncation triggers a special case that we omitted in the description of Algorithm 14, where only one prefix byte is removed instead of two[9]. We briefly discuss the case for two removed bytes at the end of this paragraph. Let $m' \leftarrow y_0 || y_1 || y_2$, where $|y_0|_8 = 1$ is the removed prefix, y_1 is the 43-byte SID returned to the adversary, and y_2 is a 212-byte unknown suffix. To recover x, we try all possible prefix values $\hat{y}_0 \in \{0,1\}^8$ and argue that for the correct prefix guess $\hat{y}_0 = y_0$, we have:

$$\Pr\left[\left\lfloor \left\lceil \frac{\hat{y}_0 || y_1 \cdot 256^{212}}{q} \right\rceil \cdot 256^{-88} \right\rfloor = u_0 || x \;\middle|\; u' < p \right] \geq 1 - 2^{-31}$$

Before explaining this probability, we observe that iterating over the 2^8 prefixes $\hat{y}_0$ is computationally feasible and does not involve the victim. For the correct prefix guess $\hat{y}_0 = y_0$, we have $y = \hat{y}_0 || y_1 \cdot 256^{212} + y_2$. We detect $\hat{y}_0$ when the result starts with u_0. This detection has a probability for false positive of approximately 2^{-64}, because $|u_0|_8 = 8$. However, we ignore false

[9] According to the source code comments, this is due to a patch for bogus legacy padding. The special case is triggered when the second leftmost byte of m' is non-zero, which happens for the roughly uniformly random value m' with probability $1 - 2^{-8}$.

positives in the further analysis, since we can recognize the correct plaintext in practice using additional information on its structure. We condition the probability on the event that $u' < p$, leading to $y = u' \cdot q$ as the previous paragraph explains. Rewriting this equation and using $u' = u_0||x \cdot 256^{88} + u_2$, we obtain:

$$\frac{\hat{y}_0||y_1 \cdot 256^{212} + y_2}{q} = u_0||x \cdot 256^{88} + u_2$$
$$\Longrightarrow \frac{\hat{y}_0||y_1 \cdot 256^{212}}{q} = u_0||x \cdot 256^{88} + u_2 - \frac{y_2}{q}$$

We bound the last term with $\frac{y_2}{q} < 2^{8 \cdot 212} \cdot 2^{-1023} = 2^{673}$. The prefix $u_0||x$ is separated by at least 31 bits[10] from $\frac{y_2}{q}$. Thus, the subtraction of $\frac{y_2}{q}$ can only affect the prefix if the bits in between are all zeros, i.e., we have the prefix $u_0||x||0^{31}$. For an approximately uniformly random value, this happens with probability 2^{-31}. Therefore, $\left\lceil \frac{\hat{y}_0||y_1 \cdot 256^{212}}{q} \right\rceil$ has the prefix $u_0||x$ with probability greater or equal to $1 - 2^{-31}$ given that $u' < p$.

In the case where $|y_0|_8 = 2$, we iterate over the 2^{16} values $\hat{y}_0 \in \{0,1\}^{16}$. We then recover the prefix $u_0||x = \left\lfloor \left\lceil \frac{\hat{y}_0||y_1 \cdot 256^{211}}{q} \right\rceil \cdot 256^{-88} \right\rfloor$ with probability $1 - 2^{-39}$, where the improved success probability comes at slightly higher computational cost.

Conclusion. Our attack recovers the decryption x of any target ciphertexts $ct_0||ct_1$, which were encrypted with `AES-ECB` under the master key k_m, with a success probability of at least $(1 - 2^{-31}) \cdot (1 - 2^{-63}) > 1 - 2^{-30}$. It only requires a single login attempt of the victim. The malicious service provider can modify the encrypted `RSA` private key, choose the encrypted session ID, and observe the SID recovered by the victim. The attack is hard to detect as a client, since the server can accept any session ID. However, the victim might observe an error message when it tries to use its garbled private key after authenticating. Then, the client likely deletes the bogus private key and refetches it. Alternatively, the service provider can also throw an authentication error and have the user re-attempt the login process.

4.3 Mega Framing Attack to Place a New File

This section presents a framing attack allowing a malicious cloud service provider to add files to a victim's cloud. An adversary can trivially *modify* existing files by recovering their keys with the `AES-ECB` decryption attack from Section 4.2 and decrypt, change, and re-encrypt the files. However, a

[10] The prefix $u_0||x$ starts after $88 * 8 = 704$ bits, the last term has less than 673 bits.

more ambitious adversary might want to preserve existing files or add more documents than the user currently stores. For instance, a conceivable attack might frame someone as a whistle-blower and place an extensive collection of internal documents in the victim's account. Such an attack might gain credibility when it preserves the target's files. Without the ability to place new files, the attacker is limited by the user's total number of files[11].

Our attack bypasses the challenge of an unavailable AES-ECB encryption oracle under the master key. Otherwise, a full chosen-message forgery would be trivial by choosing the key material and using the regular file encryption. Instead, our attack employs the decryption oracle from Section 4.2 and takes the reverse direction: it re-constructs a file for which the adversary can choose all but one AES block. We set the remaining bits such that the file encryption produces some target MAC value.

After discussing the threat model in Section 4.3.1, we describe the attack using a decryption oracle in Section 4.3.2. Section 4.3.3 explains our implementation of this attack using a MitM proxy to target Mega's web client. Furthermore, Section 4.3.4 briefly outlines how the framing attack can be performed without a full decryption oracle. Finally, we conclude in Section 4.3.5.

4.3.1 Threat Model

We consider a malicious cloud service provider who successfully performed the RSA secret key recovery attack from Section 4.1 and can use the AES-ECB decryption attack from Section 4.2. The adversary aims to place a malicious file in the victim's cloud storage, indistinguishable from genuinely uploaded files. Furthermore, we assume that the uploaded file format tolerates 128 arbitrarily chosen bits at an AES-block aligned location.

4.3.2 Framing Attack Description

On a high level, the attacker first obtains a file key k_F, file nonce N_F, and metamac M_{meta} by decrypting a randomly sampled node key ciphertext. Next, it works backward to insert a single AES block at some chosen location in the malicious file F. This block ensures that the integrity verification succeeds by producing M_{meta} when the file is encrypted with key k_F and nonce N_F. Many standard file formats such as PNG and PDF tolerate 128 injected bits (for instance, in the file's metadata, as trailing data, or in unused structural components) without affecting the displayed content. Figure 4.3 visualizes the framing attack presented in this section, where the light red blocks are constraints that the adversary must satisfy in order to pass the integrity verification.

[11] An attacker could bypass this limitation by reusing file keys for multiple documents. However, this is trivial to detect.

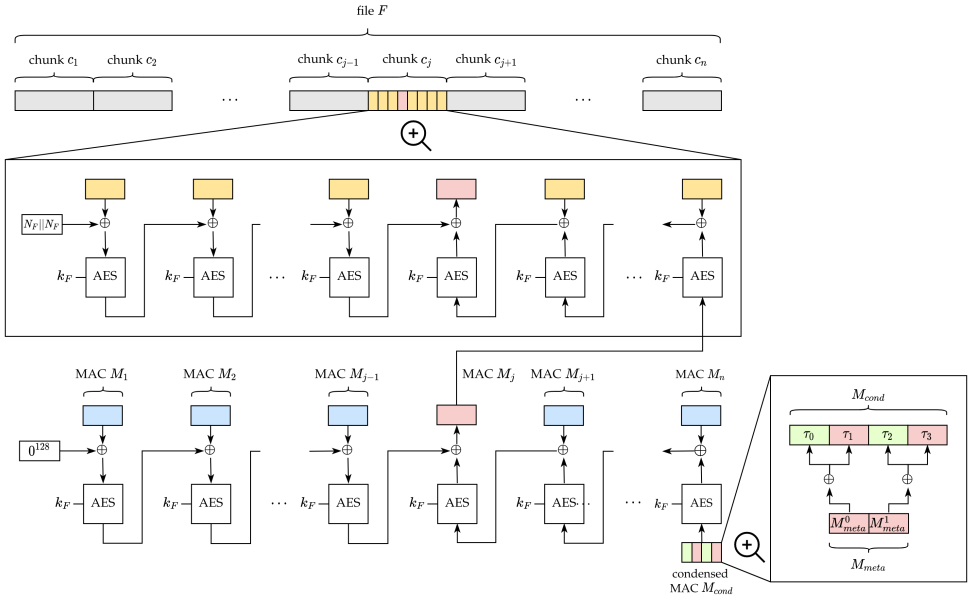

Figure 4.3: This visualization illustrates how we derive a single AES block to ensure that a predominantly chosen file produces the desired metamac M_{meta}. The light red blocks are fixed values determined by the recovered obfuscated key, all other AES blocks of the file can be chosen by the adversary. The arrows indicate the direction in which we compute values: we calculate intermediate results in the CBC-MAC before the red block as usual, but we compute blocks after the fixed one in reverse order, starting from the desired result.

Obtaining Obfuscated File Key. We select the two AES ciphertext blocks $ct_0, ct_1 \leftarrow\!\!\$\ \{0,1\}^{128}$ uniformly at random. Next, we use the AES-ECB decryption attack from Section 4.2 to recover the corresponding plaintext blocks pt_0 and pt_1. As described in Algorithm 9, an obfuscated node key consists of a 64-bit nonce N_F and the 64-bit metamac M_{meta}, concatenated to the 128-bit scrambled file key $k_F^{scr} \leftarrow k_F \oplus (N_F||M_{meta})$. Thus, the decryption gives us $k_F \leftarrow pt_0 \oplus pt_1$ and $N_F||M_{meta} \leftarrow pt_1$,. We now need to ensure that the placed file F produces the metamac M_{meta} when the victim encrypts it with key k_F and the nonce N_F.

Deriving Matching MAC. In order to construct a file with metamac M_{meta}, we work backward and select a condensed MAC M_{cond} that produces M_{meta}. For this purpose, we split the metamac into two 32-bit chunks $M^0_{meta}||M^1_{meta} \leftarrow M_{meta}$ and choose $\tau_0, \tau_2 \leftarrow\!\!\$\ \{0,1\}^{32}$ uniformly at random. Then, we set $\tau_1 \leftarrow \tau_0 \oplus M^0_{meta}$ and $\tau_3 \leftarrow \tau_2 \oplus M^1_{meta}$ and verify that the 128-bit condensed MAC

$M_{cond} \leftarrow \tau_0||\tau_1||\tau_2||\tau_3$ indeed results in our desired `metamac` M_{meta}:

$$(\tau_0 \oplus \tau_1)||(\tau_2 \oplus \tau_3) = M^0_{meta}||M^1_{meta} = M_{meta}$$

In order for our file to produce M_{cond}, we only have to select the MAC of a single chunk. We recall from Algorithm 8 that the condensed MAC is a `CBC-MAC` over the concatenation of all n chunk MACs M_i for $i \in [1, n]$. Therefore, we select $j \in [1, n]$ for which the file F tolerates 128 random bits (aligned to `AES` blocks) in the j-th chunk. Let $c_1||c_2||\ldots||c_n \leftarrow F$ be the file chunks, each consisting of l_c `AES` blocks. First, we calculate all chunk MACs except M_j using Mega's `AES-CCM` encryption:

$$\forall i \in [1, n] \setminus \{j\}.[c_i]_{k_F}, M_i \leftarrow \texttt{AES\_CCM\_ENC\_MEGA}(k_F, N_F||(i \cdot l_c), c_i)$$

We compute the partial condensed MAC $M_{cond,j-1}$ by calculating the `CBC-MAC` of $M_0||M_1||\ldots||M_{j-1}$. Moreover, we calculate intermediate condensed MAC values backward starting from the desired output $M_{cond,n} \leftarrow M_{cond}$. I.e., for $i = n-1, n-2, \ldots, j$, we calculate:

$$M_{cond,i} \leftarrow \texttt{AES-ECB.Dec}(k_F, M_{cond,i+1}) \oplus M_{i+1}$$

The remaining chunk MAC M_j is defined by the preceding value $M_{cond,j-1}$ and the intermediate condensed MAC $M_{cond,j}$ from the reverse calculation:

$$M_j \leftarrow M_{cond,j-1} \oplus \texttt{AES-ECB.Dec}(k_F, M_{cond,j})$$

Now, we have a fixed MAC M_j for the file chunk c_j. By very similar reasoning as above (except that we use the IV $N_F||N_F$ instead of 0^{128}), we can choose a single 128-bit `AES` block of the chunk, which we need to set to a chosen value such that the overall chunk MAC is M_j. We choose the chunk and `AES` block such that the 128 random bits are in a place that is either ignored by the file format or not visible to the unsuspecting user.

4.3.3 Proof of Concept Attack

Similar to Section 4.1.5, we use `mitmproxy` to realize a TLS-MitM adversary. This software allows us to spoof `TLS` certificates on the fly and impersonate Mega for the client. We implement a Python script that waits for the relevant requests and injects malicious responses. In this section, we describe the four steps performed by our implementation of the framing attack under the assumption that the adversary already recovered the victim's `RSA` private key (e.g., by using the attack from Section 4.1).

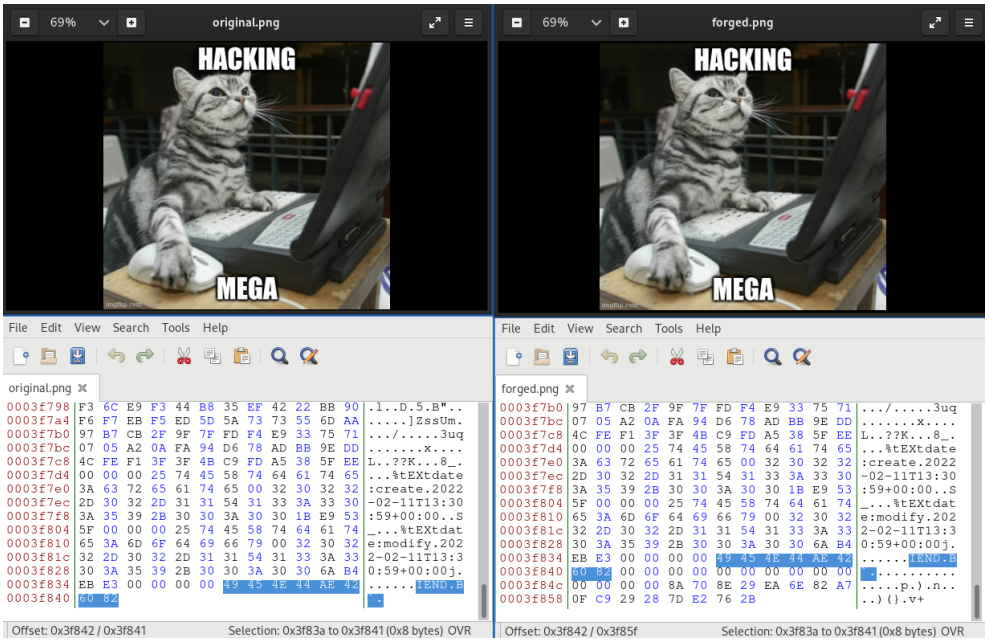

Figure 4.4: The forged image[12] (on the right) is padded with zeros (to make the image size a multiple of the AES block size) and has a trailer of 128 random bits chosen by our attack. There is no visual difference between the images.

Step 1: Obtaining an Obfuscated Node Key. We choose two random AES-ECB ciphertext blocks, ct_0 and ct_1, and perform the plaintext recovery attack described in Section 4.2. During a login attempt for a user, we inject the chosen blocks in the RSA private key ciphertext and send a maliciously chosen SID. The returned session ID allows us to recover the obfuscated key (corresponding to the decryption of ct_0 and ct_1).

Step 2: Forging a File. We rebuild Mega's AES-CCM and attribute encryption and implement the forward and backward AES-CBC calculation from Section 4.3.2 to derive a file that produces a given condensed MAC. We inject a PNG image because the format structure tolerates appended bits: a PNG starts with the magic bytes 0x89504E47 followed by a series of chunks between an IHDR and an IEND chunk. The parser ignores any data after the latter. Figure 4.4 shows that the images constructed as described in Section 4.3.2 are visually equivalent[13], but the forged file ends with 128 chosen

[12] A meme generated with https://imgflip.com/memetemplate/97924249/hacker-cat.

[13] The difference between the forged and original image is not only invisible to the naked eye: all pixels are effectively the same since the PNG parser never reads the chunk data added to the forged file.

bits. The CBC-MAC of the forged file produces M_{cond}, which gives the metamac required by the obfuscated node key from *Step 1*. From analyzing the source code, we know that Mega uses the following attribute format for the JSON key-value pairs (k_1, v_1), (k_2, v_2), etc.:

$$\texttt{MEGA}\{\texttt{"}k_1\texttt{":"}v_1\texttt{", "}k_2\texttt{":"}v_2\texttt{"}\}$$

Clients using Mega's SDK can add custom attributes, but most files contain a name (key "n") and a cyclic redundancy check (key "c"). However, we found that the latter is not required to display the file correctly in Mega's web client. A careful adversary might still want to implement the checksum to avoid deviating from genuinely inserted files. The attribute ciphertext consists of the AES-CBC encryption of the UTF-8 encoding of the above string zero-padded for the block cipher. This attribute ciphertext, together with the previous file encryption, decrypts to our forged PNG under the key from *Step 1*. Although this forged file is indistinguishable from a genuine one for Mega, we conservatively chose to perform a non-persistent attack instead of uploading self-created data. Therefore, we only inject the file on the client by manipulating server responses. Of course, a real adversary has no such ethical restrictions and will likely store the forged file persistently to avoid having to perform an active attack whenever the framed user accesses their cloud storage.

Step 3: Modifying the File Tree. We activate Mega's security setting to wipe all local metadata, which forces the client to request the file tree on login. We emphasize that this is not required for our attack in general: a malicious service provider can directly add the files to the user's cloud and pretend this is a new file uploaded by another client of the same user. The file tree fetch command contains the JSON algorithm identifier "a":"f". We intercept the server response, which contains a key "f" storing a list of file metadata. The most relevant keys present for each file are:

- "h": encoded[14] file handle (six bytes)
- "p": encoded parent handle
- "t": integer file type; important ones for us are FILENODE (0) and ROOTNODE (2)
- "a": encoded attribute encryption
- "k": encoded owner handle and obfuscated node key (separated by a colon)
- "s": file size

[14] To be precise, Mega uses URL-safe base 64 encoding for all byte values in JSON API commands.

We inject a file into the file tree by choosing a random file handle (distinct from the already existing ones) and populate the other fields honestly. We place the file in the root folder of the client's Mega storage by searching the entry in the file tree of type `ROOTNODE`. We can use this node's handle (key `"h"`) as the parent handle and the node's owner (key `"u"`) as the owner handle. The latter is necessarily the victim's handle since the root folder cannot be shared. Since we inject the obfuscated node key and attributes from *Step 2*, the client correctly decrypts them and displays our forged file in the web client. Extended file attributes (key `"fa"`) could contain handles for additional data such as thumbnails. A motivated adversary can forge this part as well, but we limit our proof of concept to the file content.

Step 4: Serving the Forged File. When the victim double clicks on the injected file, Mega's in-browser image viewer opens, and the client requests a file with the command `"a":"g"` and our chosen file handle (stored with key `"n"` in the command). Instead of forwarding this request, we directly impersonate the server's response (since Mega's server does not store any file for our injected handle). A valid response provides the client with information to fetch the file from Mega's user storage (which is a different server than the one handling API requests) and contains the following fields:

- `"g"`: one or multiple URL(s) of the requested data on
- `"ip"`: a list of IPv4 and IPv6 addresses for every URL in `"g"`
- `"s"`: file size
- `"at"`: encoded attribute encryption
- `"pfa"`: permission settings for attributes

We choose a valid Mega domain and a random path for the URL to avoid issues with the browser's Content Security Policy (CSP). We set the other values consistent with the file tree metadata and choose `"pfa"` to allow the client to write attributes. As a result of this injected answer, the client requests the file chunk-by-chunk from the sent URL, where it specifies the start and end values in the path. We again intercept this request and serve the encrypted file from *Step 2* as a binary transfer (using the response content-type `application/octet-stream`). Since we constructed the file to decrypt correctly under the key from *Step 1* and produce the correct `metamac`, the client displays the forged image in the victim's cloud.

4.3.4 Framing Attack Without Decryption Oracle

Although Section 4.2 implements a decryption oracle, this is a rather strong prerequisite in general, and, even in the case of Mega, the initial key recovery in Section 4.1 required for the decryption attack is intricate. However, we

can perform the framing attack with a weaker adversary by trading off stealthiness. Given at least two known plaintext-ciphertext blocks of AES-ECB under the master key k_M (i.e., without relying on an AES-ECB decryption oracle), the adversary can use them as node key ciphertexts and perform, as described in Section 4.3.2, the reverse derivation of a mostly chosen file F that produces a fixed condensed MAC. Using such known blocks restricts the number of derivable unique node keys. Thus, if we insert more documents than we have node keys, repeated key ciphertexts indicate the attack since they are very unlikely to occur during genuine file encryption.

4.3.5 Conclusion

In conclusion, the above attack allows a malicious cloud provider to place any file F in the user's cloud as long as the file format tolerates 128 random AES-block aligned bits. Using an AES decryption oracle, the forged file is not suspicious because the reversely constructed node key is indistinguishable from genuine node keys due to AES-ECB being a pseudo-random permutation. Therefore, the framed user lacks cryptographic arguments to convince a third party that it did not upload F. A weaker adversary can perform the framing attack only with known plaintext-ciphertext pairs, which either limits the number of forged documents or makes the attack easier to detect due to repeated key ciphertext blocks. This framing attack is made possible by the lack of integrity protection for the key ciphertexts.

4.4 Bleichenbacher Variant With Unknown Prefix

In this section, we discuss a new variant of Bleichenbacher's attack [88] to decrypt RSA ciphertexts by using a padding oracle. Instead of targeting PKCS#1 v1.5, we extend the attack to Mega's custom padding scheme involving an unknown prefix. Although this attack is weaker than the RSA key recovery from Section 4.1, it does not require the ability to overwrite keys. Furthermore, it attacks a different instance of RSA encryption and can be performed by a slightly weaker adversary. Section 4.4.1 explains the oracle exposed by Mega's code for decrypting chat keys with RSA. Next, Section 4.4.3 discusses our guess-and-purge strategy that accounts for an unknown prefix while keeping the number of additional queries reasonable. Finally, Section 4.4.6 concludes with an empirical performance analysis.

4.4.1 Oracle in Mega's Legacy Chat Key Decryption

Algorithm 17 describes the decryption of chat key material using the client's private RSA key sk_{share}[15]. The client parses the decrypted value m_{bytes} as

[15] Algorithm 4 specifies Mega's generation of the RSA key pair (sk_{share}, pk_{share}).

four components: a prefix y of two bytes, a two-byte length encoding L, concatenated keys K, and some random padding r to make m_{bytes} of the size of the RSA modulus. The length encoding function l (as introduced in Section 4.1) transforms an integer to a two byte big-endian encoding. The inverse l^{-1} recovers the byte length of the plaintext from the length encoding L. The length validation on Line 4 of Algorithm 17 checks that the decrypted and unpadded K consists of key-sized blocks. A user can send multiple chat keys concatenated to a single message. The client aborts if K cannot be split into chat keys, i.e., $|K|_8$ is not a multiple of the chat key length. The decryption tolerates an encoded length that exceeds the number of available bytes due to the error resilience of the web client's JavaScript implementation. In that case, the plaintext has maximal size, i.e., $|r|_8 = 0$. Although such an invalid L does not raise an error nor result in an out-of-bound read, the length validation on Line 4 can still fail if the maximal size is not a multiple of 16.

Under the assumption that the adversary can observe the error symbol returned by Line 5, one might get the impression that this construction only leaks whether $\lfloor l^{-1}(L) \rfloor_{16} = 0$. However, upon closer inspection, there is more leakage depending on the size of the RSA modulus N. The maximal plaintext length is $|N|_8 - 4$ because the encoding uses four bytes for the prefix and length encoding. Since $|N|_8$ is a power of two larger than sixteen, we know that $\lfloor |N|_8 - 4 \rfloor_{16} = 12$. Therefore, the length validation fails for all values $l^{-1}(L) \geq |N|_8 - 4$ because they all result in K of length $|K|_8 = |N|_8 - 4$, which does not pass Line 4 of Algorithm 17.

In summary, we know for a successful decryption that $l^{-1}(L) < |N|_8 - 4$ and $\lfloor l^{-1}(L) \rfloor_{16} = 0$. Concretely, for RSA-2048 we decrypt successfully iff L is of the form $0^8||b_3b_2b_1b_0||0^4$ with four unknown bits b_i for $i \in [0, 3]$. Our variant of the Bleichenbacher attack described in Section 4.4.3 mainly uses the zero prefix. Using the rightmost four zero bits of the length encoding could enable further optimizations.

4.4.2 Threat Model

We consider a malicious cloud provider or an adversary impersonating Mega. Unlike in the previous attack, the real cloud provider is not strictly required to perform the attack, since no tampering with encrypted key material sent by the server is necessary. Given a target ciphertext, this attack can be performed offline using a direct connection between victim and adversary. More importantly, integrity protecting the key encryption does not protect against this attack.

Algorithm 17 Mega's RSA decryption of chat key ciphertext c_{key} containing one or more symmetric AES keys.

procedure MEGA_RSA_DECRYPT_CHAT_KEYS(c_{key}, (sk_{share}, pk_{share}))
  $m_{bytes} \leftarrow \texttt{RSA.Dec}(sk_{share}, c_{key})$ ▷ m_{bytes} is a $|N|_8$-byte string
  $y||L||K||r \leftarrow m_{bytes}$
  ▷ Where $|y|_8 = 2$ (ignored prefix), $|L|_8 = 2$ (length encoding), $|K|_8 = \min(l^{-1}(L), |N|_8 - 4)$ (keys), and r (random padding)

  ▷ Length validation
  if $\lfloor |K|_8 \rfloor_{16} \neq 0$ **then**
    return $\perp$
  return K

4.4.3 Guess-And-Purge Variant of Bleichenbacher's Attack

This section explains our modifications of the original Bleichenbacher attack steps [88] to account for Mega's leakage pattern and the unknown prefix. Section 4.4.4 discusses practical optimizations, Section 4.4.5 provides the mathematical reasoning for our modifications, and Section 4.4.6 evaluates the performance empirically. On a high level, our attack tries to *guess* the unknown prefix y and removes it from the result before performing Bleichenbacher's attack adapted to Mega's length encoding leakage. We quickly identify and *purge* wrong guesses with subsequent multipliers.

To stay close to the notation of [88], let $c = \lfloor m^e \rfloor_N$ be the RSA ciphertext (with modulus N and public exponent e) of a target message m. Let $B \leftarrow 256^{|N|_8-4}$ be the power of two that exceeds the largest possible unencoded plaintext by one[16]. Let m_0 be the conforming multiple of the target plaintext. If m_0 is of known length, then the maximal plaintext interval is $[|m_0|_8 \cdot B, (|m_0|_8 + 1) \cdot B - 1]$. We introduce the notation of workloads $w = (\mathcal{M}_{i,t_i}, \mathcal{H}_{i,t_i})$, where $\mathcal{M}_{i,t_i}$ is the set of closed intervals after iteration i, resulting from the choice of multipliers $\mathcal{H}_{i,t_i} = (s_{0,t_0}, s_{1,t_1}, \ldots, s_{i,t_i})$ with the guesses $t_j \in \{0,1\}^{16}$ for all $j \in [0,i]$ of the most significant two bytes of $\texttt{lpad}_{|N|_8}\left(\left\lfloor s_{j,t_j} \cdot m_0 \right\rfloor_N\right)$ (where $\texttt{lpad}_N$ left pads $\left\lfloor s_{j,t_j} \cdot m_0 \right\rfloor_N$ with zero bytes to $|N|_8$ bytes). We denote with $t_0 = t_1 = \varepsilon$ that there is no prefix guess for the first two multipliers. As explained in detail below, they are chosen independent of any prefix guess: *Step 1* chooses s_{0,t_0} randomly and *Step 2.a* linearly searches for s_{1,t_1}, starting from a value derived from the initial bounds. For $k \in [2,i]$, *Step 2.c* chooses the multiplier $s_{k,t_k} \in \mathbb{Z}$ based on the shifted prefix guess $y_k \leftarrow 256^2 \cdot B \cdot t_k$ of the conforming message

[16]The maximal plaintext byte length is $|N|_8 - 4$ because there are always two unknown prefix bytes and two length bytes.

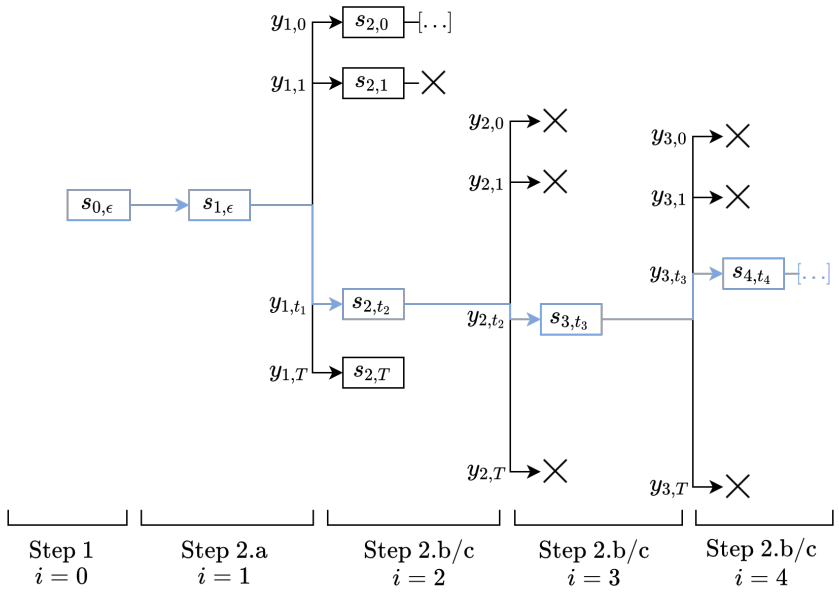

Figure 4.5: Visualization of the guess-and-purge strategy for our modified Bleichenbacher attack with $T = 2^{16}$ prefix guesses $t \in \{0,1\}^{16}$. The blue path up to iteration i corresponds to a workload $w = (\mathcal{M}_{i,t_i}, \mathcal{H}_{i,t_i})$, where $\mathcal{M}_{i,t_i}$ is the implicitly associated solution interval that depends on the previous choice of multipliers $(s_{0,\varepsilon}, s_{1,\varepsilon}, s_{2,t_2}, \ldots, s_{i,t_i})$ stored in $\mathcal{H}_{i,t_i}$. We no longer follow paths with empty intervals (marked by a cross).

$\texttt{lpad}_{|N|_8}\left(\left\lfloor s_{k,t_k} \cdot m_0 \right\rfloor_N\right)$[17] and the previous intervals $\mathcal{M}_{k-1,t_{k-1}}$ such that s_{k,t_k} reduces the size of the possible solution intervals adequately. We guess the prefix of $\texttt{lpad}_{|N|_8}\left(\left\lfloor s_{k,t_k} \cdot m_0 \right\rfloor_N\right)$ before selecting s_{k,t_k}. The attack tree visualization of Figure 4.5 shows that the indices t_k define a path by specifying prefix guesses for a workload. The indices t_k differ for every workload and if we guessed all previous prefix values y_j $\forall j \in [1,k)$ correctly, then we have $m_0 \in [a,b]$ for $[a,b] \in \mathcal{M}_{k,t_k}$. Otherwise, we quickly arrive at a contradiction with $\mathcal{M}_{k,t_k} = \emptyset$ and eliminate this workload. Furthermore, we introduce the oracle $\mathcal{O}_c(s)$ described in Algorithm 18, which returns true iff the `RSA` decryption from Algorithm 17 succeeds for the target ciphertext c multiplied by s^e modulo N.

For our modified Bleichenbacher attack, we perform *Step 1* once at the beginning of the attack and repeat *Step 2* to *Step 4* for every workload $w = (\mathcal{M}_{i-1,t_{i-1}}, \mathcal{H}_{i-1,t_{i-1}}) \in \mathcal{W}$, where i is counting the iterations through all steps.

[17] In other words, $\left\lfloor s_{k,t_k} \cdot m_0 \right\rfloor_N - y_k$ is in the plaintext interval defined by the length encoding, where we can make use of the zero bits in the prefix similar to Bleichenbacher's attack on the leading `0x0002` of `PKCS#1 v1.5`.

Algorithm 18 Oracle checking whether the padding format required for our modified Bleichenbacher attack is satisfied when decrypting $\lfloor s^e \cdot c \rfloor_N$ for a target ciphertext c.

1: **procedure** $\mathcal{O}_c(s)$
2: $c' \leftarrow \lfloor s^e \cdot c \rfloor_N$
3: $m \leftarrow$ `mega_rsa_decrypt_chat_keys`$(c', (sk_{share}, pk_{share}))$
4: **return** $m \neq \perp$

Step 1: Blinding. Given a target ciphertext $c = \lfloor m^e \rfloor_N$, we sample random multipliers $s_{0,\varepsilon}$ until $\mathcal{O}_c(s_{0,\varepsilon})$ returns true. For the first successful value $s_{0,\varepsilon}$, we set:

$$\begin{aligned} c_0 &\leftarrow \lfloor c \cdot (s_{0,\varepsilon})^e \rfloor_N \\ \mathcal{M}_{0,\varepsilon} &\leftarrow \{[pt_{min}, pt_{max}]\} \\ \mathcal{H}_{0,\varepsilon} &\leftarrow (s_{0,\varepsilon}) \\ \mathcal{W} &\leftarrow \{(\mathcal{M}_{0,\varepsilon}, \mathcal{H}_{0,\varepsilon})\} \\ i &\leftarrow 1 \end{aligned}$$

where $pt_{min} \leftarrow 0$ is the smallest possible plaintext and $pt_{max} \leftarrow (|N|_8 - 15) \cdot B - 1$ is the largest possible plaintext (including length encoding), because $|N|_8 - 16$ is the first conforming length[18] that is smaller than $|N|_8 - 4$, and $B - 1$ is the largest possible content of that length[19]. The subsequent attack recovers $m_0 \leftarrow \lfloor s_{0,\varepsilon} \cdot m \rfloor_N$, which is the decryption of c_0. We do not need any prefix guess (as indicated by ε) as $\mathcal{M}_{0,\varepsilon}$ contains a single interval specifying the maximum range of conforming plaintexts.

Step 2: Searching for s satisfying $\mathcal{O}_c(s)$.

Step 2.a: Starting the search. For $i = 1$, we have only a single workload with $\mathcal{M}_{0,\varepsilon} = \{[a, b]\}$[20]. We search for the smallest $s_{1,\varepsilon} \geq N/(b+1)$ such that $\mathcal{O}_c(s_{1,\varepsilon})$ is true.

Step 2.b: Sequential searching with $|\mathcal{M}_{i-1,t_{i-1}}| > 1$. For $i > 1$ and more than one interval left, where $s_{i-1,t_{i-1}} \in \mathcal{H}_{i-1,t_{i-1}}$, we search for the smallest $s_{i,t_i} > s_{i-1,t_{i-1}}$ where $\mathcal{O}_c(s_{i,t_i})$ is true.

Step 2.c: Interval-based searching with $|\mathcal{M}_{i-1,t_{i-1}}| = 1$. For $i > 1$ and exactly one interval $[a, b]$ left, where $s_{i-1,t_{i-1}} \in$

[18] Recall from Algorithm 17 that the length must be a multiple of 16 and we assume $|N|_8$ to be a multiple of 16.

[19] Thus, $(|N|_8 - 16) \cdot B + B - 1 = (|N|_8 - 15) \cdot B - 1$ is the largest message with the length encoding $|N|_8 - 16$.

[20] In the generic case, $a = 0$ and $b = (|N|_8 - 15) \cdot B - 1$.

$\mathcal{H}_{i-1,t_{i-1}}$, we iterate over the prefix guesses $t_i \in [0, 2^{16})$ with the corresponding shifted values $y_i \leftarrow 256^2 \cdot B \cdot t_i$ of the yet to be computed value $\texttt{lpad}_{|N|_8}\left(\lfloor s_{i,t_i} \cdot m \rfloor_N\right)$. We search the smallest pair of variables r_i and s_{i,t_i} that satisfies the following two constraints as well as $\mathcal{O}_c(s_{i,t_i})$. Due to the choice of r_i and s_{i,t_i}, we approximately halve the interval $[a, b]$ in *Step 3* (see Section 4.4.5).

We start incrementing r_i from

$$r_i \geq \frac{2 \cdot b \cdot s_{i-1,t_{i-1}} - pt_{min} - y_i}{N}$$

For every r_i value, we try the following multipliers:

$$\frac{pt_{min} + r_i \cdot N + y_i}{b} \leq s_{i,t_i} < \frac{pt_{max} + r_i \cdot N + y_i}{a}$$

Section 4.4.5 discusses our reasoning for this search procedure in detail. However, the intuition is as follows: there exists at least one solution because we are guaranteed to find a conforming s_{i,t_i} value for the workload with all correct prefix guesses since our procedure then performs *Step 2.c* from the original Bleichenbacher attack without an unknown prefix. If we end up using another multiplier s_{i,t_i} for a wrong prefix guess, this still reduces our intervals. Although this s_{i,t_i} might not eliminate as many plaintext candidates as the one for the correct prefix, it is still a correct multiplier because the oracle decision is independent of our prefix guess.

Step 3: Narrowing the set of solutions. For all $[a, b] \in \mathcal{M}_{i-1,t_{i-1}}$ and $\mathcal{H}_{i-1,t_{i-1}} = (s_{0,t_0}, s_{1,t_1}, \ldots, s_{i-1,t_{i-1}})$, we update the bounds for every prefix guess $t^* \in \{0,1\}^{16}$ of $\texttt{lpad}_{|N|_8}\left(\lfloor s_{i,t_i} \cdot m \rfloor_N\right)$ and the corresponding $y^* \leftarrow 256^2 \cdot B \cdot t^*$, as follows:

$$\begin{aligned}
\mathcal{M}_{i,t^*} \leftarrow & \cup_{a,b,r} \left\{ \left[a', b'\right] \right\} \\
& \text{for } a' \leftarrow \max\left(a, \left\lceil \frac{pt_{min} + r \cdot N + y^*}{s_{i,t_i}} \right\rceil\right) \\
& \text{and } b' \leftarrow \min\left(b, \left\lfloor \frac{pt_{max} - 1 + r \cdot N + y^*}{s_{i,t_i}} \right\rfloor\right) \\
& \text{where } \frac{a \cdot s_{i,t_i} - pt_{max} + 1 - y^*}{N} \leq r \leq \frac{b \cdot s_{i,t_i} - pt_{min} - y^*}{N} \\
\mathcal{H}_{i,t^*} \leftarrow & \left(s_{0,t_0}, s_{1,t_1}, \ldots, s_{i-1,t_{i-1}}, s_{i,t^*}\right), \text{ for } s_{i,t^*} \leftarrow s_{i,t_i}
\end{aligned}$$

We add a new workload $(\mathcal{M}_{i,t^*}, \mathcal{H}_{i,t^*})$ to $\mathcal{W}$ whenever $\mathcal{M}_{i,t^*} \neq \emptyset$.

It is necessary to consider all possible prefixes $t^* \in \{0,1\}^{16}$ to guarantee the existence of a fully correct prefix guess history $t_0, t_1, \ldots, t^*$ (which is implicitly stored in $\mathcal{H}_{i,t^*}$). For instance, if we would only use the prefix t_i for which *Step 2.c* found the multiplier s_{i,t_i}, m_0 might not be in any of the remaining intervals because the first two bytes t^* of $\texttt{lpad}_{|N|_8}\left(\lfloor s_{i,t_i} \cdot m_0 \rfloor_N\right)$ are not equal to t_i.

Step 4: Computing the solution. If there is only one workload $\mathcal{W} = \{(\mathcal{M}_{i-1,t_{i-1}}, \mathcal{H}_{i-1,t_{i-1}})\}$ left and only one interval $\mathcal{M}_{i-1,t_{i-1}} = \{[a,a]\}$ containing a single value, we have $a = m_0$ and return the solution $m \leftarrow \left\lfloor a \cdot (s_0)^{-1} \right\rfloor_N$. Otherwise, if there is no workload for iteration i left (i.e., $\forall(\mathcal{M}_{i^*,t_{i^*}}, \mathcal{H}_{i^*,t_{i^*}}) \in \mathcal{W}.\, i^* > i$), we set $i \leftarrow i+1$ and continue with *Step 2*. If there is still a workload left for iteration i, we go to *Step 2* with that one.

4.4.4 Optimizations

Dynamic Programming to Avoid Duplicate Oracle Queries. We introduce the set $\mathcal{S}$ of already queried multiplier intervals and maintain the following invariant: $\forall I \in \mathcal{S}. \forall s \in I.\, \neg\mathcal{O}_c(s)$. To query the s values in $[s_{min}, s_{max}]$ in *Step 2*, we consider the intervals $\{[a_0,b_0), [a_1,b_1), \ldots, [a_n,b_n)\} \subseteq \mathcal{S}$ such that the intervals are ordered by increasing bounds $\forall i \in [0,n).\, b_i < a_{i+1}$ and $\forall i \in [0,n]\,.\, [a_i,b_i) \cap [s_{min}, s_{max}] \neq \emptyset$. Then, we only query s values from the following intervals (i.e., $s \notin \cup_{S \in \mathcal{S}} S$) until we find a conforming value s^*:

$$[s_{min}, a_0), [b_0, a_1), [b_1, a_2), \ldots, [b_{n-1}, a_n), [b_n, s_{max})$$

The first and last intervals are empty if $s_{min} \in [a_0, b_0)$ respectively $s_{max} \in [a_n, b_n)$. After every query, we update $\mathcal{S}$ to reflect the new queries: $\mathcal{S} \leftarrow \mathcal{S} \cup \{[\min(a_0, s_{min}), s^*)\}$. We implement $\mathcal{S}$ as a sorted list and merge overlapping intervals to improve performance. This enables us to conduct efficient binary searches for the intervals of new query values s.

Optimizing Step 1 by Specialization. We can often start with a significantly shorter interval in *Step 1*, given more knowledge on the initial ciphertext. If we decrypt a single 16B chat key, we already know that $\mathcal{O}_c(1)$ is satisfied because the given ciphertext c encrypts a padding-conforming message. Consequently, we can skip the random search for $s_{0,\varepsilon}$. Furthermore, we know the plaintext length is 16 and, thus, can reduce the initial interval to $\mathcal{M}_{0,\varepsilon} \leftarrow \{[16 \cdot B, 17 \cdot B - 1]\}$.

Optimizing Step 4 by Specialization. We can optimize *Step 4* in case the target message m was already conforming (i.e., we skipped *Step 1*), and there

are γ bits of random padding in the LSBs of m. Since we do not need to recover the padding, we can abort as soon as the last interval $\mathcal{M}_{i,t_i} = \{[a, b]\}$ has a stable prefix containing all non-padding bits, i.e., $\lfloor a \cdot 2^{-\gamma} \rfloor = \lfloor b \cdot 2^{-\gamma} \rfloor$. The larger the random padding is, the earlier we can stop the attack. In comparison, the classic Bleichenbacher attack is forced to recover the entire padding because `PKCS#1 v1.5` left pads the message. In other words, the message is in the least significant bits and therefore requires a remaining interval of size one for recovery. With Mega's padding scheme, we can terminate early because the least significant bits are padding, and, thus, a larger interval can contain only a single message with different paddings.

Skipping Holes and Trimming. We evaluated the applicability of advanced optimization techniques for Bleichenbacher's attack presented by Bardou et al. [81]. The *skipping holes* optimization improves the lower bound for $s_{1,\varepsilon}$ in *Step 2.a* by observing that we require $s_{1,\varepsilon} \cdot m_0 > pt_{min} + N$ such that $\mathcal{O}_c(s_{1,\varepsilon} \cdot m_0)$ holds for $s_{1,\varepsilon} \cdot m_0 \geq N$. Therefore, we improve the lower bound to $s_{1,\varepsilon} \geq \frac{pt_{min}+N}{pt_{max}}$ using $m_0 \leq pt_{max}$, which saves us approximately $\frac{pt_{min}}{pt_{max}}$ queries. Such *holes* of non-conforming multipliers can further be identified in *Step 3*. However, as analyzed in [81], we can only skip significant regions together with the *trimming* optimization. *Trimming* searches for factors of m_0 and uses them to trim the search interval. Unfortunately, proposition 1 of [81][21] cannot be adapted to detect factors in our setting due to the unknown prefix.[22] Further work could search for a condition to detect factors of m_0 in our setting, which would enable *trimming* and increase the benefit of *skipping holes*.

4.4.5 Analysis of the Introduced Modifications

Bleichenbacher's attack [88] builds on the following observation for some multiplier s_i:

$$s_i \cdot m_0 \text{ conforming } \implies \exists r_i \in \mathbb{Z}.\ 2 \cdot B' \leq s_i \cdot m_0 - r_i \cdot N < 3 \cdot B'$$

where $B' \leftarrow 256^{|N|_8 - 2}$ because a valid `PKCS#1 v1.5` padding starts with `0x0002`. The r_i value removes the reduction modulo N of $\lfloor s_i \cdot m_0 \rfloor_N$. Bleichenbacher's steps provide a structured approach to finding s_i values that significantly reduce the interval of possible plaintext values.

[21] Proposition 1 of [81] (for the classic Bleichenbacher attack): Let u and t be two coprime positive integers such that $u < \frac{3}{2}t$ and $t < \frac{2N}{9B}$. If m_0 and $m_0 \cdot ut^{-1} \mod N$ are `PKCS#1 v1.5` conforming, then m_0 is divisible by t.

[22] The core issue is that Bardou et al. can bound the size of conforming value $x \leftarrow m_0 \cdot ut^{-1} \mod N < 3B$ since `PKCS#1 v1.5` requires them to start with the prefix `0x0002`. Therefore, they have $xt < N$ since t is a small factor, and can derive $xt = m_0 u \implies t \mid m_0$. In our setting, x has an unknown prefix leading to $xt > N$ even for very small factors t with high probability.

The challenge in our scenario is that conforming messages start with an unknown prefix y_i. We adapt the above observation to account for y_i as follows:

$$\mathcal{O}_c(s_i) \Longrightarrow \exists r \in \mathbb{Z}, t^* \in \{0,1\}^{16}. \\ pt_{min} \leq s_i \cdot m_0 - r \cdot N - 256^2 \cdot B \cdot t^* \leq pt_{max} - 1 \tag{4.1}$$

Let r and $t^* \in \{0,1\}^{16}$ with the corresponding shifted value $y^* \leftarrow 256^2 \cdot B \cdot t^*$ be values satisfying the right-hand side of Equation (4.1) for some s_i with $\mathcal{O}_c(s_i)$. We solve the inequalities for m_0 to derive the bounds for m_0 used in *Step 3* to narrow intervals:

$$\frac{pt_{min} + r \cdot N + y^*}{s_i} \leq m_0 \leq \frac{pt_{max} - 1 + r \cdot N + y^*}{s_i} \tag{4.2}$$

Furthermore, we can derive the bounds for r in *Step 3* from Equation (4.1) by using $a \leq m_0 \leq b$ for some interval $[a,b] \in \mathcal{M}_{i,t^*}$ and $m_0 \in [a,b]$:

$$\begin{aligned} \frac{s_i \cdot a - pt_{max} + 1 - y^*}{N} \leq \frac{s_i \cdot m_0 - pt_{max} + 1 - y^*}{N} \leq r \\ r \leq \frac{s_i \cdot m_0 - pt_{min} - y^*}{N} \leq \frac{s_i \cdot b - pt_{min} - y^*}{N} \end{aligned} \tag{4.3}$$

Very similarly, we derive the bounds for s_i in *Step 2* using $\frac{1}{b} \leq \frac{1}{m_0} \leq \frac{1}{a}$ for some interval $[a,b] \in \mathcal{M}_{i,t^*}$ and $m_0 \in [a,b]$:

$$\frac{pt_{min} + r \cdot N + y^*}{b} \leq s_i < \frac{pt_{max} + r \cdot N + y^*}{a} \tag{4.4}$$

Correctness. We can use the above statements to prove the correctness of our algorithm by induction over i. Let $T(i) \equiv \exists t_i \in \{0,1\}^{16}.\ \exists\, [a,b] \in \mathcal{M}_{i,t_i}.\ m_0 \in [a,b]$ be our induction hypothesis stating that in every iteration, there exists at least one interval containing the target message m_0. Since the last interval has length one, this implies that we find the correct plaintext m_0 and, thus, return the decryption m of the target ciphertext c. The base case $T(0)$ trivially holds because $\mathcal{M}_{0,\varepsilon} = \{[pt_{min}, pt_{max}]\}$ and $m_0 \in [pt_{min}, pt_{max}]$ by definition since pt_{min} and pt_{max} are the smallest, respectively largest, plaintext values. We assume $T(i-1)$ for the induction step and show $T(i)$. *Step 2* uses some s_i with $\mathcal{O}_c(s_i)$ by construction. Therefore, by Equation (4.1) there exist r and $t^* \in \{0,1\}^{16}$ such that the right-hand side equality holds. By Equation (4.3), we know that the range of r values used in *Step 3* contains the correct one. Furthermore, we iterate over all $t^* \in \{0,1\}^{16}$ and add intervals to $\mathcal{M}_{i,t^*}$. Therefore, for the correct r and t^*, we narrow $[a,b]$ to $[a',b']$ in *Step 3* where the bounds from Equation (4.2) guarantee that $m_0 \in [a',b']$. We conclude the induction proof by noting that $[a',b'] \in \mathcal{M}_{i,t^*}$ implies $T(i)$.

Interval Reduction. We motivate the choice of the lower bound of r_i in *Step 2.c* by showing that it leads to roughly halved intervals. Let s_i be the multiplier chosen in *Step 2.c* satisfying $\mathcal{O}_c(s_i)$ and let y^* be the prefix of $\texttt{1pad}_{|N|_8}(\lfloor s_i \cdot m_0 \rfloor_N)$. We recall from *Step 2.c* for the previous multiplier s_{i-1} of this workload:

$$r_i \geq \frac{2 \cdot b \cdot s_{i-1} - pt_{min} - y^*}{N} \tag{4.5}$$

We deviate from Bleichenbacher's original *Step 2.c* in addition to subtracting the prefix by moving the factor 2 in front of $b \cdot s_{i-1}$ to enable the reasoning below.

Step 3 updates the bounds of the interval $[a, b]$ to $[a', b']$, where:

$$a' \leftarrow \max\left(a, \left\lceil \frac{pt_{min} + r_i \cdot N + y^*}{s_i} \right\rceil\right)$$
$$b' \leftarrow \min\left(b, \left\lfloor \frac{pt_{max} - 1 + r_i \cdot N + y^*}{s_i} \right\rfloor\right)$$

We now show that the interval length $|[a', b']|$ is approximately half of $|[a, b]|$. First, we use Equation (4.4) and Equation (4.5) to derive a lower bound for s_i:

$$s_i \geq \frac{pt_{min} + r_i \cdot N + y^*}{b} \geq 2 \cdot s_{i-1}$$

Consequently, we derive the following for the interval bounds:

$$\begin{aligned} \left|\left[a', b'\right]\right| &\leq \frac{pt_{max} - 1 + r_i \cdot N + y^*}{s_i} - \frac{pt_{min} + r_i \cdot N + y^*}{s_i} \\ &\leq \frac{pt_{max} - 1 - pt_{min}}{s_i} \\ &\leq \frac{pt_{max} - 1 - pt_{min}}{2 \cdot s_{i-1}} \end{aligned}$$

The last interval size is approximately half of the maximal interval size of $|[a, b]|$.

4.4.6 Empirical Evaluation

This section evaluates the performance of our modified Bleichenbacher attack and shows that wrong prefix guesses quickly result in empty intervals. We target the `RSA`-2048 encryption of a single 16-byte chat key using Mega's custom padding scheme.

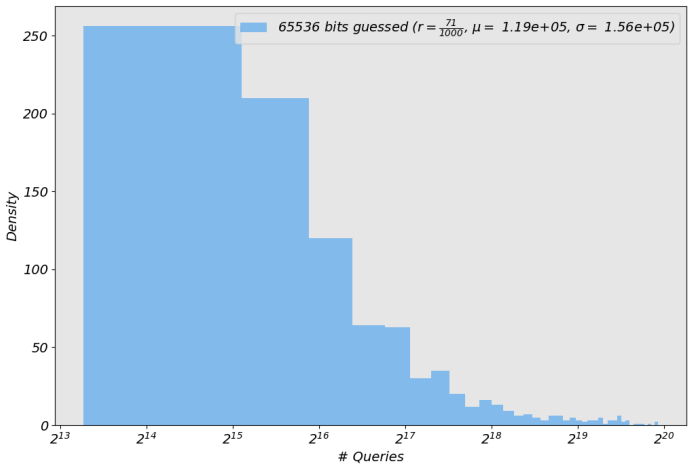

Figure 4.6: Density of the number of oracle queries (calls to $\mathcal{O}_c$) required by our variant of Bleichenbacher's attack when recovering a 16-byte chat key encrypted with RSA-2048. There is a 2-byte unknown prefix, and we aborted 71 out of 1000 runs because they exceeded the threshold of 10^6 queries.

Query Complexity. The density histogram in Figure 4.6 shows that our attack has a query complexity of approximately $2^{16.9}$ on average with a comparatively high standard deviation. A quarter of all runs only require 2^{14} queries, but the distribution has a long tail, and we aborted 71 out of 1000 runs because they exceeded our cutoff of 10^6 queries. We use the Freedman-Diaconis binning rule [106] to decide for an appropriate number of bins of equal width.

Workload Purging. The query complexity of our attack is significantly lower than executing 2^{16} classic Bleichenbacher attacks for every prefix guess. Figure 4.7 visualizes the core reason: every conforming multiplier s_{i,t_i} allows us to detect workloads with an invalid prefix guess in $\mathcal{H}_{i,t_i}$ because they result in an empty solution interval $\mathcal{M}_{i,t_i} = \emptyset$. The stacked bar plot shows that the first multiplier $s_{1,\varepsilon}$ adds approximately 2500 plausible prefix guesses. The next multiplier s_{2,t_2} eliminates more than 95% of the wrong guesses shown with a blue bar in Figure 4.7; the remaining workloads are gray with an error bar showing the standard deviation. As the solution intervals narrow, every multiplier adds fewer new workloads while following multipliers rapidly eliminate wrong guesses. After 28 multipliers, only a single workload remains, and (in most cases) a multiplier leads to only one valid prefix guess. Therefore, we do not require more queries than the classic Bleichenbacher attack after this point. In general, we require a fraction of the multipliers

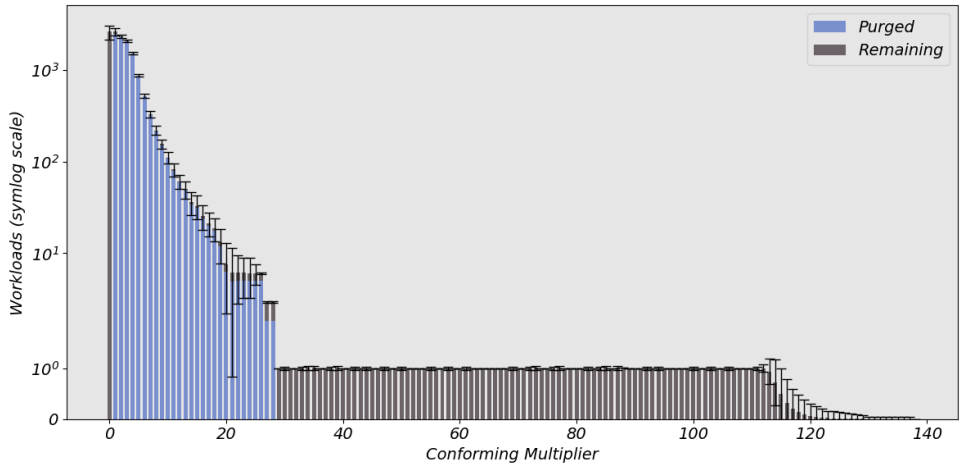

Figure 4.7: Number of purged and remaining workloads for every multiplier s_{i,t_i} satisfying $\mathcal{O}_c(s_{i,t_i})$. The error bars show the standard deviation of the purged workloads across 929 runs (1000 runs, 71 of them aborted).

necessary for attacking `PKCS#1 v1.5` since we do not need to recover the padding. The total number of multipliers ranges from 110 to 140, which is one reason for the long tail in Figure 4.6. However, finding the first multipliers often affects the number of oracle queries more significantly.

Guessing Padding Bits for PKCS#1 v1.5. This paragraph applies the prefix guessing Bleichenbacher attack on `PKCS#1 v1.5`. Figure 4.8 shows that the direct application requires more queries when guessing padding bits than the same code (which approximates the classic Bleichenbacher attack with minor improvements) needs without guessing. We see in Figure 4.9 that *Step 2.a* is executed more when we guess padding bits. The attack on Mega does not have this behavior because we know the initial interval and guess prefixes of all $s_{i,t_i} \cdot m \mod N$ values. In contrast, guessing padding bits means adding multiple initial intervals in *Step 1* (one for every padding guess) and then executing *Step 2.a* for every workload. Although we detect most wrong padding guesses with a single multiplier, we still execute the expensive *Step 2.a* for the remaining ones. Bardou et al. [81] introduced the *Skipping Holes* and *Trimming* optimizations for *Step 2.a*. Although Section 4.4.4 discussed the difficulties of adapting them for our attack in the Mega setting, we could implement them when guessing `PKCS#1 v1.5` padding bits. We observe that *Trimming* has a similar effect as padding guessing: it reduces the initial interval by finding small factors t of m_0. On the one hand, this process has the benefit to reduce intervals without causing additional executions of *Step 2.a*. On the other hand, our variant of Bleichenbacher's attack does not require searching optimal factors to reduce the interval, it is only limited

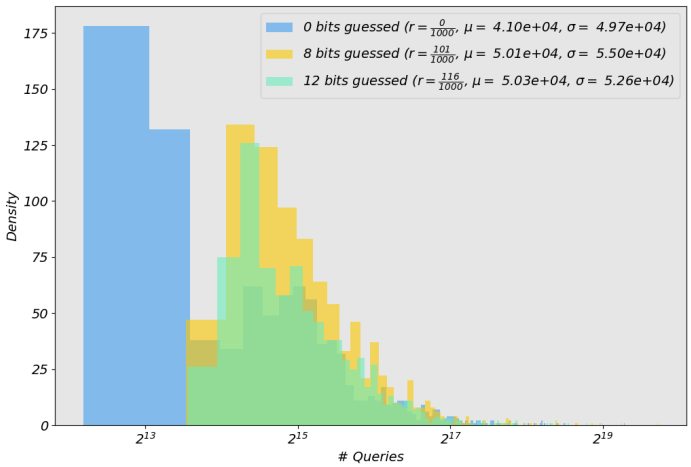

Figure 4.8: Densities for the number of oracle queries (calls to $\mathcal{O}_c$) required when applying our Bleichenbacher variant to guess the first 0, 8, or 12 bits after the `0x0002` prefix of the `PKCS#1 v1.5` padding. The ratio r represents the fraction of aborted runs (exceeding 10^6 queries) over all runs, μ is the average number of queries and σ the standard deviation.

by the additional computational cost of guessed prefixes and the queries necessary to eliminate wrong guesses.

Future work could evaluate whether there is a beneficial combination of padding-guessing and *Trimming* to avoid the cost of *Step 2.a*. Furthermore, our approach of maintaining intervals and purging empty ones could be extended to account for noisy oracles. Bleichenbacher's attack already tolerates when an oracle fails to recognize a correctly padded message (false negative) at the cost of more queries. However, classifying a message as `PKCS#1 v1.5` conforming by mistake can cause Bleichenbacher's attack to fail. Such false positives are challenging when building oracles from microarchitectural side-channels [152]. Our modified version could be extended with backtracking when no valid solution remains. Since our variant quickly detects wrong intervals given a correct multiplier, it would be interesting to compare its performance to repeated measurements proposed by Ronen et al. [152].

4.4.7 Conclusion

This section contributes a new variant of Bleichenbacher's attack, which can tolerate unknown prefixes. We evaluated this variant to require $2^{16.9}$ queries on average despite guessing a two-byte prefix. The main reason for a performance in the same order of magnitude as the original Bleichenbacher

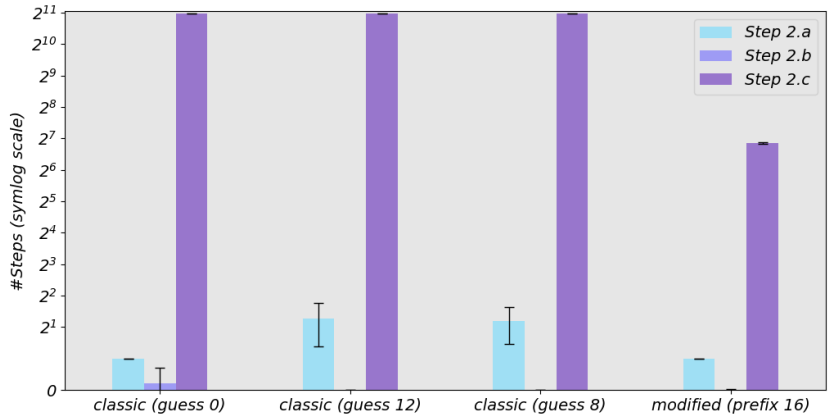

Figure 4.9: Number of times an attack executes *Step 2.a*, *Step 2.b*, or *Step 2.c* from Section 4.4.3. We compare four variants: attacking `PKCS#1 v1.5` and guessing the 0, 8, or 12 leftmost bits of the padding, and our attack on Mega's padding scheme, guessing a two-byte unknown prefix.

attack is that we can use multipliers for wrong prefix guesses to purge incorrect guesses and quickly identify the correct prefix.

This attack is challenging to exploit in practice for multiple reasons: first, it requires a substantial amount of queries. Second, the error oracle is challenging to realize because the legacy `RSA` decryption is only a fallback if no `Curve25519` keys are available[23]. Our static analysis indicates that the web client reports two different error messages to the server depending on whether Algorithm 17 failed to decrypt the keys (i.e., $\mathcal{O}_c(s)$ is false) or successfully recovered some invalid keys that will later fail on a chat message decryption (i.e., $\mathcal{O}_c(s)$ is true). We did not attempt to implement this oracle in practice due to the high reverse-engineering effort required to reconstruct the server code and trigger this legacy code.

Despite the theoretical nature of the attack in this section, it still points out two weaknesses in Mega's design beyond the key overwriting attacks from Section 4.1 and Section 4.2. First, it is dangerous to use a custom `RSA` padding scheme. Although we need a non-trivial adaption of Bleichenbacher's attack to exploit it, the cryptographic literature agrees that the non-linear and provable `RSA-OAEP` padding scheme is preferable. Second, the reuse of a single `RSA` key pair for decrypting shared node keys as well as chat keys allows an adversary to attack the former using deprecated legacy code in the latter code path. Although the format of shared keys is not compatible with

[23] According to source code comments, accounts created before 2016 may still execute this code even without further malicious operations of the service provider [18].

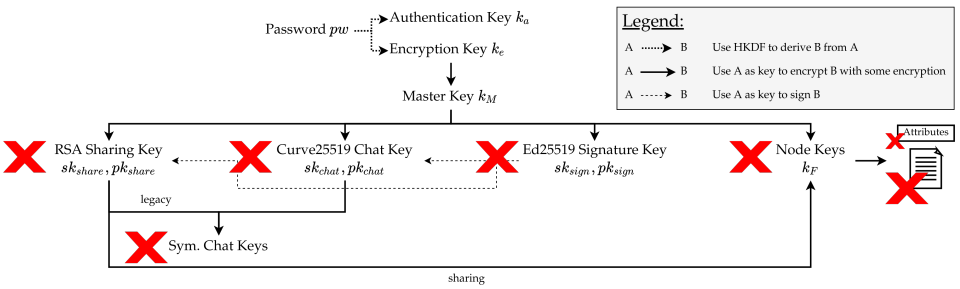

Figure 4.10: Overview of Mega's key hierarchy and the parts affected by our attacks.

the one of chat keys, *Step 1* finds a multiplier $s_{0,\varepsilon}$ such that $\mathcal{O}_c(s_{0,\varepsilon})$ is satisfied. It is feasible to find this multiplier because the message only requires twelve bits to be zero[24], which happens approximately with probability 2^{-12} for multipliers chosen uniformly at random.

4.5 Conclusion

Figure 4.10 shows Mega's key hierarchy and marks the parts affected by our attacks with red crosses. This chapter discussed an attack to recover the `RSA` sharing key in 512 user login attempts by combining key overwriting, observations on `RSA-CRT`, and lattice optimizations. Building on this attack, an adversary can recover all other keys encrypted with the master key k_M using our `AES-ECB` plaintext recovery attack. The symmetric chat keys are also affected because they are exchanged using compromised asymmetric key material. Furthermore, an attacker can decrypt node keys and break the confidentiality of files and attributes. Additionally, our framing attack enables the adversary to place new files in a user's cloud storage, indistinguishable from a genuinely uploaded file. We present a new variant of Bleichenbacher's attack on `PKCS#1 v1.5` applied to Mega's custom padding scheme, which accounts for an unknown prefix using a novel Guess-and-Purge strategy.

Despite being able to decrypt everything encrypted with the master key, we have not compromised this key itself. Furthermore, our attacks do not compromise the user password. However, Section 3.3.10 outlined how any entity knowing the authentication key k_a (including Mega) can perform dictionary attacks on passwords with low entropy.

[24]Messages with the structure $0^8||b||0^4$ for some $b \in \{0,1\}^4$ are valid.

Chapter 5

Mitigation of Attacks on Mega

This section describes countermeasures that Mega can employ to address the attacks from Chapter 4. The scale of Mega poses many practical challenges to changing their cryptographic design fundamentally. For instance, modifying the file encryption procedure requires re-encrypting over 1000 petabytes of user data. In addition to the high computational cost, this also causes massive traffic: users need to download all their data, decrypt it with the legacy file encryption, re-encrypt it with the new procedure, and then upload it again. Mega reports to have a total uplink capacity of more than 1000 Gbit/s [23]. Even at constant maximal utilization, it would take approximately 185 days to re-encrypt all data with this bandwidth. The large size can also be an issue for customers since their device needs to do expensive cryptographic operations for re-encrypting the user's file due to end-to-end encryption. For instance, a mobile phone might be quite overwhelmed to re-encrypt 500 GB of data. Furthermore, backward compatibility is a significant issue: it is not realistic that all 243 million users of Mega would re-encrypt their entire storage within reasonable time due to various reasons (inactive account, lost password, unreachable user). Therefore, Mega must either continue to support the current design for backward compatibility or render some customer's data inaccessible after a grace period. The former approach brings the danger of downgrade attacks on users who have already transitioned to the new design, and the second implicates the loss of customers. Due to these significant practical challenges, we organize our mitigation in three levels: first, Section 5.1 describes immediate countermeasures that can be implemented during the responsible disclosure period and prevent the most severe security breaches through our attacks. Second, Section 5.2 describes more substantial changes that address our attacks in more detail while still avoiding expensive changes to the design (such as file re-encryption). Finally, Section 5.3 outlines long-term goals for redesigning the cryptographic architecture and addressing the root causes of our attacks. Figure 5.4 provides

an overview of how the key hierarchy would change based on the different mitigation steps.

This section does not aim to provide complete mitigation of Mega's services. They have multiple clients with a considerable code base, and the cryptographic primitives are used at different locations to provide various functionality. Despite not analyzing everything, we point out fundamental issues in the current design, most of which we exploited with the attacks presented in Chapter 4.

5.1 Immediate Countermeasures

The immediate countermeasures that need to be in place to avoid putting user data at significant risk due to the attacks we discovered are the following: adding ad hoc integrity protection for key ciphertexts (Section 5.1.1), key separation (Section 5.1.2), and enforcing a stricter `RSA` padding format (Section 5.1.3). Figure 5.1 shows the key hierarchy after applying the immediate countermeasures. We remark that these measures do *not* mitigate all of our attacks, nor do they address the root flaws in the cryptographic design. Instead, they are a set of measures that we consider practical to deploy during a 90-day disclosure window, even for data of the scale that Mega governs. Some of our proposed measures are temporary and should be replaced as soon as possible. We recommend to directly implement the extensions discussed in Section 5.2 whenever a more thorough redesign is feasible. Moreover, Section 5.3 suggests long-term mitigation goals, which may require a significant redesign of Mega's cryptographic architecture and may be challenging to deploy in practice.

5.1.1 Ad Hoc Integrity Protection for Key Ciphertexts and File Attributes

The missing integrity protection of key ciphertexts enables the malicious cloud provider or TLS-MitM adversary to perform key overwriting attacks. Our `RSA` key recovery from Section 4.1 exploits this attack vector during authentication to factor the `RSA` key. As part of the most immediate countermeasures, we propose the suboptimal solution to add integrity protection on top of the existing encryption by using `HMAC` on the `AES-ECB` key ciphertexts. Section 5.1.2 discusses how to derive keys to compute the `HMAC` tags.

Furthermore, we propose adding integrity protection to the `AES-CBC` encryption of the attributes. Although our attacks do not specifically target the attributes, changing them could confuse the client. For instance, it is possible to construct "binary polyglots" [77], which are files that are valid in two file formats, and an adversary tampering with the attributes could decide which of them is displayed. More theoretically, the file encryption trivially fails to

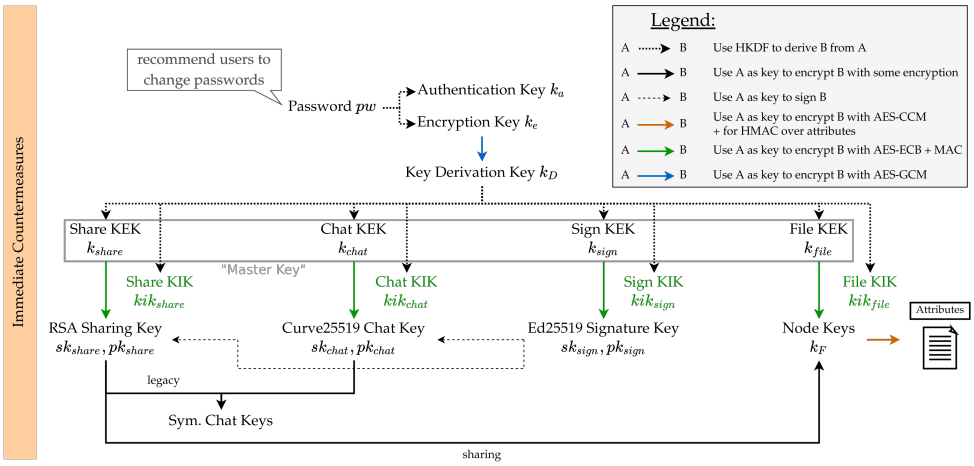

Figure 5.1: Redesign of the current key hierarchy for the immediate countermeasures.

achieve existential unforgeability as long as the integrity of attributes is not protected: an adversary can change the encrypted attributes and, thereby, produce a valid ciphertext.

This solution is unsatisfactory in multiple regards. First, implementing encryption and `HMAC` has various pitfalls, and even large projects such as the Google Keyczar cryptographic library have made severe mistakes in the past [133]. For instance, Nate Lawson showed that non-constant time `HMAC` comparison leads to forgery attacks [133]. Furthermore, accidentally implementing MAC-then-Encrypt or Encrypt-and-MAC instead of Encrypt-then-MAC likely leads to padding oracles or might leak information. Second, although we add integrity protection, the keys are still encrypted with `AES-ECB`. Therefore, our construction does not achieve authenticated encryption security since the deterministic `AES-ECB` is not `IND-CPA` secure. Therefore, better but more invasive mitigations proposed in Sections 5.2 and 5.3 are to switch to a modern authenticated encryption scheme, such as `AES-GCM`, which transparently includes efficient integrity protection. Using existing implementations in widely used cryptographic libraries lowers the threat of introducing vulnerabilities such as side-channel attacks.

Nevertheless, we decided to propose this suboptimal solution due to its ease of implementation. Since we only extend the existing encryption, older clients can remain functional and simply ignore the new authentication tags. While not addressing the root causes properly, careful implementation of this measure suffices to prevent our key recovery attack (Section 4.1) and complicate the `AES-ECB` decryption and framing attacks (Sections 4.2 and 4.3).

5.1.2 Master Key Separation

The *Key Separation Principle* is a widely accepted best practice in cryptography, which [108] summarizes for block ciphers as "one should always use distinct keys for distinct algorithms and distinct modes of operation." Mega violates this principle on multiple occasions. The most immediate violation is that they use the same node key for both `AES-CCM` encrypting file chunks and `CBC-MAC` in order to produce chunk MACs and the `metamac`. Furthermore, Mega uses the same `RSA` key pair to share files, receive the SID during authentication, and exchange chat keys in a legacy solution. Unfortunately, both violations are intricate and challenging to address without requiring expensive operations such as file re-encryption. For this reason, we defer mitigating these key reuses to Sections 5.2 and 5.3.

However, a more pressing issue is the reuse of the master key k_M: although not used in different primitives, it is protecting the `RSA` sharing key, the `Curve25519` chat key, the `Ed25519` signature key, and all node keys. Our master key decryption attack from Section 4.2 exploits this key reuse to decrypt all key material using an attack vector on the `RSA` key. We propose to replace the master key with different keys *derived* from a new, uniformly at random chosen key derivation key k_D. The client encrypts k_D using an AEAD-scheme such as `AES-GCM` under the encryption key k_e while specifying the key's purpose in the associated data, and uploads this key ciphertext to Mega. As Figure 5.1 shows, we propose to use `HKDF` [128] as a key derivation function (KDF) to derive different *key encryption keys* (KEKs) to encrypt Share, Chat, Sign, and Node Keys. As recommended in section 3.1 of RFC 5869 [129], we advise to use a salt that is stored together with the encrypted k_D (either encrypted itself or as associated data) to strengthen `HKDF`.

Since the mitigation from Section 5.1.1 introduced MACs for every key ciphertext, we additionally need to derive a *key integrity key* (KIK) for each of these MACs to avoid reusing the same key for `AES-ECB` encryption and `HMAC` computation. We avoid deriving k_D directly from the password to support password recovery: users can export k_D and securely store it offline. If they forget their password, they can reset it and encrypt k_D locally with the new encryption key derived from the updated password. We remark that it is not necessary to re-encrypt other key material on a password change, since k_D remains unchanged. However, as discussed in Section 5.3.3, a desirable extension is to add support for key rotation nonetheless.

For the `HMAC` tag used in the attribute encryption, it is challenging to do proper key separation without affecting the 1000 petabytes of legacy encrypted data or introducing the possibility for downgrade attacks. Ideally, and as part of the recommended measures presented in Section 5.3, we derive different keys for attribute and file encryption. However, since we need to derive them from the per-file key k_F, this changes the key used to encrypt files.

Consequently, users need to re-encrypt their files, or the clients need to support both encryptions simultaneously, which adds significant complexity. For this reason, we tolerate *violating the Key Separation Principle* as part of the short-term mitigations and reuse k_F for the HMAC tag. We argue that the benefit of integrity protection through HMAC outweighs the issues introduced by reusing k_F. The existing key reuse in AES-ECB, AES-CBC, and CBC-MAC leads to potentially worse interactions than the one introduced by HMAC, since the AES primitives all use the same block cipher. However, we stress that this is a purely heuristic argument without provable evidence. This construction should only be accepted as a temporary solution due to the urgency of the vulnerabilities and Mega's scale.

We consider this change feasible because clients can lazily transition to the new hierarchy. Essentially, whenever a key is used, a client updated with the improved key hierarchy can first use k_M to decrypt the key, re-encrypt it with the appropriate KEK, and upload the new key ciphertext to the server. Legacy clients will continue to receive the key material encrypted under the master key since they have not updated the key encryptions yet. Therefore, they will continue to work until Mega deprecates older and insecure clients. A deployment challenge is that as soon as one device of a user updated the key material, their clients on other devices have to be updated as well to be able to decrypt the new key material. Consequently, updates on different platforms have to be coordinated to not lock the user out of his account on devices with outdated clients. Furthermore, we want to avoid adding more legacy code and supporting both key encryptions simultaneously since this could lead to downgrade attacks.

Mega should recommend their users to update their password as a proactive protection measure to render the old master key k_M inaccessible. Without changing the password, the encryption key k_e remains the same and can still decrypt k_M. Thus, if a user can be tricked to decrypt the master key ciphertext (for instance, by using an outdated client), our attacks can still be performed by a malicious cloud provider who stored the superseded key ciphertexts. Nevertheless, even users who do not update their password benefit from the key separation, as all files uploaded with a new client can no longer be compromised by our attacks. In addition, performing a downgrade attack and storing all outdated key material increases the difficulty of performing our attacks.

5.1.3 Stricter RSA Padding Format

We propose to enforce stricter client-side checks of Mega's padding to increase the number of queries for our Bleichenbacher-style RSA decryption attack from Section 4.4. Again, this is a very short-term and heuristic measure and does not remove the padding oracle. The further countermeasures in

Sections 5.2 and 5.3 transition to RSA-OAEP, which adequately addresses the malleability of RSA ciphertexts.

We recall Mega's custom RSA padding has the structure $y||L||K||r$, where y is an unused two-byte prefix, L is a length encoding of 2 bytes, K are the padded keys, and r is some randomness. As a temporary solution, we propose to set y to a constant value C and verify it on the client. Therefore, conforming plaintexts for RSA-2048 need to satisfy $y = C$ in addition to $L = 0^8||b||0^4$ for some $b \in \{0,1\}^4$. This reduces the probability of finding a multiplier that satisfies the oracle. As an approximation, we consider $y||L||K||r \leftarrow \{0,1\}^{2048}$ where $|y|_8 = |L|_8 = 2$. In that case, $\Pr[L = 0^8||b||0^4] = 2^{-12}$ and $\Pr[L = 0^8||b||0^4 \wedge y = C] = 2^{-28}$, i.e., we expect the number of queries of our attack to increase by a factor of 2^{16} to approximately 2^{33} queries.

This "countermeasure" is certainly not a long-term solution, and an attack requiring 2^{33} queries is still worrisome as optimizations are conceivable. Nevertheless, this measure reduces the practicality of Bleichenbacher-style attack and makes it easier to detect. Furthermore, this should be a minimal and backward compatible change because Mega can choose the y value that we expect some clients to require for legacy reasons.

5.2 Minimal Countermeasures

This section presents the minimal countermeasures to address all of our attacks in a pragmatic manner. We still refrain from proposing measures that require a fundamental redesign connected with costly migration. Consequently, this section does not resolve all root causes for our attacks. However, we switch to more standard cryptographic primitives when it does not involve the re-encryption of large data volumes. In particular, this means that we ideally need to deprecate old clients or, alternatively, support new and old client versions in parallel and carefully design protocols to avoid downgrade attacks on users who updated all their clients and affected metadata. The following two measures either build on Section 5.1 or replace previous countermeasures. Section 5.2.1 replaces the temporary AES-ECB and HMAC construction with AES-GCM to encrypt keys. Section 5.2.2 proposes using two separate RSA keys to share node keys and to exchange legacy chat keys. Figure 5.2 reflects the changes introduced by our minimal countermeasures.

5.2.1 AEAD for Key Ciphertexts

This section replaces the ad hoc AES-ECB and HMAC construction to encrypt the share, chat, and sign keys with AES-GCM, an authenticated encryption scheme standardized in NIST SP 800-38D [101]. This substitution has the advantages of avoiding the implementation pitfalls discussed in Section 5.1.1 and replacing the IND-CPA-insecure AES-ECB. Furthermore, it removes the

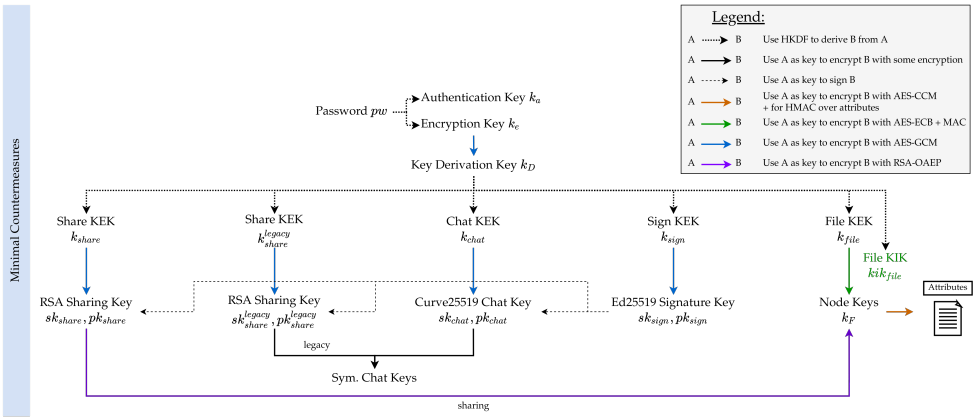

Figure 5.2: Redesign of the current key hierarchy for the minimal countermeasures.

necessity for KIKs for all except the node keys since `AES-GCM` internally takes care of encryption and integrity protection. We advise Mega to use the key's purpose and, for asymmetric primitives, the public key as associated data to avoid key confusion attacks. This countermeasure appropriately addresses the key overwriting issue raised and exploited in multiple of our attacks. However, `AES-GCM` needs to be implemented carefully to avoid issues with nonce reuse [121], cache side-channel attacks [114, 124], and fragile authentication [104, 110]. We recommend using a well-tested library and adhering to NIST SP 800-38D [101].

We defer applying this improvement for node keys to Section 5.3 because many keys would require re-encryption. A pragmatic solution might be to perform this change as part of the migration of the entire file encryption since that already requires changing the node keys. However, we expect this to be feasible for key ciphertexts since there are only three keys per user. A practical challenge of this mitigation is backward compatibility. For instance, the same user might have both new and old clients on different devices. Therefore, Mega either needs to force the user to update their clients, which requires the update to be available on all platforms, or Mega needs to continue storing the legacy key ciphertexts, which might enable downgrade attacks.

5.2.2 RSA-OAEP for RSA Sharing Keys

We recommend adding an `RSA` key pair to separate the two uses of `RSA` encryption for sharing node keys and transferring chat keys. For this purpose, we introduce $(sk_{share}^{legacy}, pk_{share}^{legacy})$ exclusively for the legacy chat key exchange. In this legacy code, we preserve Mega's custom padding since we assume

this code exists because it is challenging to change on some clients. This code is still vulnerable to our variant of Bleichenbacher's attack. However, the key separation at least prevents this insecure legacy code from affecting other encryptions. For sharing node keys, we use a different key pair (sk_{share}, pk_{share}) and RSA-OAEP. Bellare and Rogaway proposed this scheme [82], and it was standardized in PKCS#1 v2.1 [120]. RSA-OAEP was shown to be IND-CCA-secure in the random oracle model under the standard RSA assumption. Although this reduction is not very tight [92], RSA-OAEP is widely adopted and considered to be non-malleable in practice. Therefore, it fundamentally thwarts Bleichenbacher-style attacks on the padding since we cannot find multipliers to create modified ciphertexts that would be accepted and whose parsing could leak information.

We remark that this mitigation preserves the SID exchange, which Section 4.1 abuses as a partial decryption oracle. However, this is no longer a security concern due to the newly added integrity protection of the RSA secret key: patched clients should only respond with the correct session ID (which is already known to the malicious cloud provider) or an error upon tampering. Nevertheless, the guarantees that Mega obtains from encrypting the SID instead of sending it in plaintext over the TLS connection remain unclear. We hypothesize that Mega wants to force clients to demonstrate access to the RSA secret key since this implies a functional key hierarchy and access to the master key[1]. Depending on the unknown server implementation, this can be advantageous because Mega could avoid allocating state for malfunctioning (third-party) clients or adversaries who leaked the authentication key but not the user password.

5.3 Recommended Measures

In this last set of measures, we discuss long-term goals for a cryptographic refactoring of Mega's architecture to adequately address all of our attacks and protect against future ones by replacing non-standard primitives and cryptographic constructions with provably secure ones. The measures proposed in this section no longer retain backward compatible but knowingly insecure functionality. For instance, we remove the legacy chat key transfer. Since we formulate long-term goals, it is reasonable to assume that Mega can deprecate legacy clients at some point and force users to update. However, apart from being disruptive, our measures stay in line with the current cryptographic architecture and propose a secure way to implement it without fundamental changes to the design or functionality. Furthermore, we focus on achieving confidentiality and integrity and refrain from aiming to achieve

[1] The client derives the master key from the user password and uses it to decrypt all key material, including the RSA secret key.

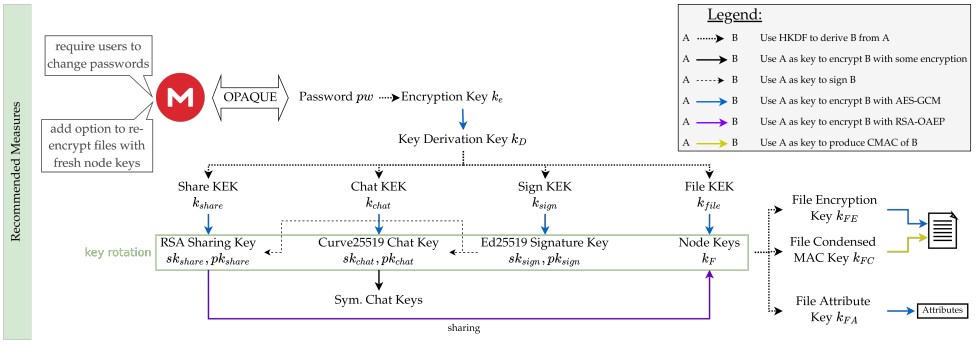

Figure 5.3: Redesign of the current key hierarchy for the recommended measures.

more advanced properties like perfect forward secrecy or post compromise security. Section 5.3.1 introduces a new file encryption with proper key separation, and Section 5.3.2 replaces the authentication procedure with an augmented PAKE. Finally, Section 5.3.3 discusses an extension of the current design that we consider beneficial to add. Figure 5.3 visualizes the effect of all our changes on the key hierarchy.

5.3.1 Refactoring Node Encryption

This section finally replaces the `AES-ECB+HMAC` construction for node key encryption with `AES-GCM`. Furthermore, we derive separate keys from the node key for different purposes and replace the flawed `AES-CCM` encryption of file chunks.

Node Keys. We replace the ad hoc node key encryption with `AES-GCM` for the same reasons as for the other keys (see Section 5.2.1). This requires changing all key ciphertexts. However, the subsequent changes modify the file encryption keys themselves and, thus, re-encrypting them is required in any case. Furthermore, we advise not to shorten the condensed MAC M_{cond} to half its size as part of the `metamac` in the obfuscated file key. Although we could not exploit this part due to the unavailable chunk MACs and the encrypted obfuscated file key, we would not be surprised if there would be a $\mathcal{O}(2^{64})$ attack since the size reduction means 2^{64} different M_{cond} result in the same `metamac`.

Key Separation. Instead of using the node keys directly, we use `HKDF` to derive three keys: a file encryption key k_{FE} to encrypt file chunks with `AES-GCM`, a chunk MAC condensing key k_{FC} to aggregate the chunk authentication tags into the condensed MAC M_{cond}, and a file attribute encryption key k_{FA}. This separation finally removes the extensive reuse of the same key for

AES-CCM encrypting chunks, aggregating the chunk MACs with CBC-MAC, and AES-CBC encryption for the attributes. Our ad hoc integrity protection from Section 5.1.1 added yet another key reuse for HMAC in the absence of a better non-invasive solution.

File Encryption. Since changing the key material requires re-encryption, we can take this opportunity to change the used algorithms. First, we replace the flawed implementation of AES-CCM[2] with the more widely adopted and efficient AES-GCM. We must be careful not to reuse IVs for AES-GCM under the same key [121]. However, this only requires us to ensure that all file chunks have different IVs since every file has a different key. A possible implementation uses a counter that starts at zero or a random value. The advantage of the former is that the IV does not need to be stored or derived. The latter improves the security guarantees since it reduces the probability of nonce reuse occurring across all users and files. We can still avoid storing an IV for every file by deriving it from the node key or storing one randomly chosen value per user as part of the account metadata. The following paragraph discusses our proposal for changing the metamac computation. Furthermore, we replace the AES-CBC+HMAC construction for attributes with AES-GCM too.

Metamac. AES-GCM, as specified in NIST SP 800-38D [101], returns a ciphertext and an authentication tag for every chunk. The absence of an integrity-protected specification of the total number of chunks requires us to take measures to prevent truncation attacks. We note that the previous approach with counters for the IVs already avoids reordering attacks. While other and arguably more elegant designs exist[3], we stay close to Mega's current approach and compute a metamac over the chunk MACs. However, we follow the recommendation of NIST SP 800-38B [102] and replace textbook CBC-MAC with CMAC [87] because the latter supports authenticating variable-length messages. We could not perform truncation attacks on Mega's CBC-MAC implementation because individual chunk MACs are not available. However, Mega's developer documentation states that checking the integrity of partial reads is a planned future enhancement [48]. This addition could add a forgery attack if full intermediate chunk MACs are published. To avoid such subtle pitfalls, we replace the use of CBC-MAC on the concatenation of a variable number of chunk MACs with CMAC. The computed metamac prevents an adversary from omitting chunks from the file end.

[2]Section 3.3.9 discussed that Mega does not encrypt the authentication tag, as demanded by the standard [164], and, therefore, implements an Encrypt-and-MAC scheme instead of MAC-then-Encrypt.

[3]For instance, the AWS Encryption SDK provides a protocol and message format specifications for framed data transfer.

The proposed measures in this section require users to re-encrypt their files because the key material is derived differently, and we use different primitives. As discussed earlier, re-encrypting over 1000 petabytes of data bears significant practical challenges and requires a long mitigation window to give customers time to transition. While this needs to be planned carefully, the long-term goal should be to replace insecure legacy code and temporary patches. The previously proposed minimal solutions are only temporary suggestions that are not unlikely to be vulnerable to more attacks due to the extensive key reuse and non-standard combination of primitives.

5.3.2 Augmented PAKE for Authentication

We briefly discussed in Section 3.3.10 that Mega's unusual authentication is susceptible to dictionary attacks by the malicious cloud provider or a TLS-MitM adversary. Since the client transmits the authentication key k_a, which is half of the output of PBKDF2, an adversary knowing k_a and the salt can try different passwords. Although publishing some output of PBKDF2 is not a violation of the PKCS#5 v2.1 [144] standard, [100] showed that such attacks on PBKDF2 are feasible on custom hardware and GPUs.

We propose to replace the current authentication with an augmented Password Authenticated Key Exchange (augmented PAKE). In such a protocol, the client proves knowledge of the password to the server without the latter needing to store password-equivalent data. Concretely, we propose to use OPAQUE [118] recommended by the Crypto Forum Research Group (CFRG). OPAQUE won the CFRG's PAKE selection process [97], which was held at the request of the Internet Engineering Task Force (IETF). In addition to not leaking password-related information during authentication, OPAQUE also provides security against pre-computation attacks [118]. This construction removes the need for an authentication key: a user can directly authenticate using their password. We only need to derive the key encryption key k_e from the password, which is used to wrap the key derivation key k_D. We use k_D and HKDF to derive further keys for various purposes. Figure 5.3 shows the restructuring of the authentication.

After implementing OPAQUE, security-conscious users should change their password since an adversary could have stored k_a from a previous authentication and therefore perform a dictionary attack once the user becomes a worthwhile target. To avoid being vulnerable due to recorded information, users should refrain from using their Mega password from this point onward for any account since it might have been compromised and linked to their email address.

5.3.3 Recommended Extension

As an extension of Mega's current design, we consider key rotation for share, chat, sign, and node keys to provide security benefits at a manageable effort. We can include expiration dates in the *associated data* of `AES-GCM` encryption. Since there is significantly less key material than data, we consider it feasible for client devices to rotate keys regularly. Key rotation of public keys limits the number of messages encrypted with that key and, thus, reduces the impact of compromising a single key. Moreover, built-in functionality to rotate keys makes responding to future security incidents more manageable.

We note that rotating the KEKs requires changing the key derivation key k_D since the KEKs are derived from k_D. Support for changing k_D might be an additional, desirable property since a malicious cloud provider could store old encryptions of k_D and decrypt it once the adversary learns a leaked password. In that case, even if the user already changed to a new password, the same key derivation key is still used to protect all keys and learning k_D enables the adversary to decrypt all keys and compromise the data they protect. To achieve a notion of forward secrecy against compromises of past passwords, Mega would need to change k_D on password changes and re-encrypt all keys with the new KEKs.

5.4 Conclusion

Figure 5.4 summarizes the effect of all countermeasures on the key hierarchy. The intermediate and minimal measures sometimes lead to a more complex structure due to temporary and backward-compatible solutions. However, the design in the recommended measures does not increase complexity and is fundamentally more secure than the current architecture. It adds key separation, integrity protection for key material and attributes, secure user authentication, and replaces non-standard primitives. Together, these improvements prevent key overwriting attacks, padding oracles, and pre-computation or dictionary-based attacks on the password. They protect the confidentiality and integrity of the user's data, key material, and identity.

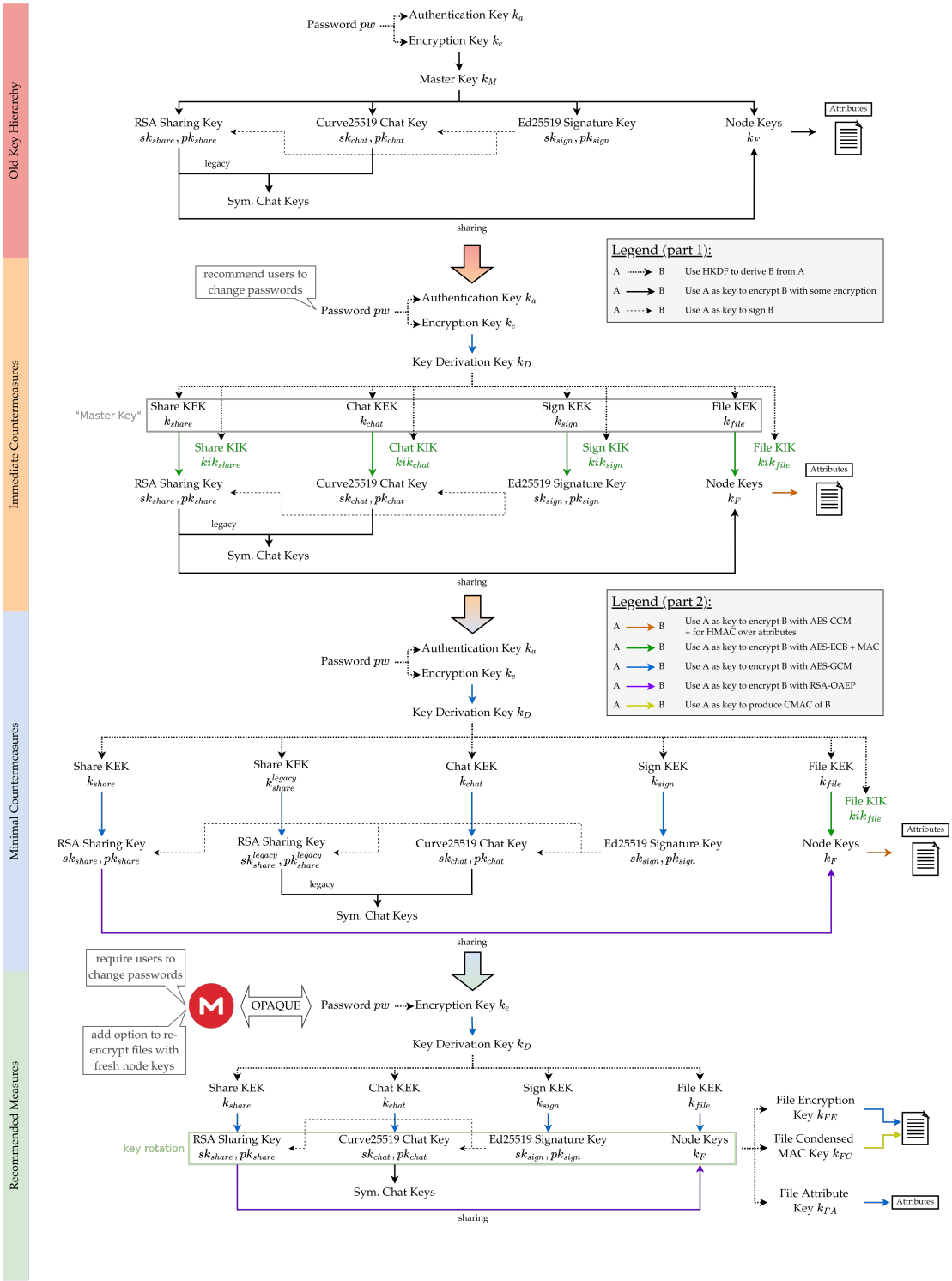

Figure 5.4: Redesign of the current key hierarchy for the immediate, minimal, and recommended countermeasures.

Chapter 6

Discussion of the Status Quo and Future Work

We discuss our principal conclusions from analyzing Mega and hypothesize about root causes for cryptographic flaws in practice. In an attempt to generalize our insights to any cryptographic design operating on a large scale, we examine how cryptographic architectures aiming to achieve standard security notions fail in practice (Section 6.1), what the consequences of these incidents are (Section 6.2), and why they occur (Section 6.3). In our opinion, there is a gap between the cryptographic community and industry, leading to some well-analyzed solutions not being adopted. We advocate in Section 6.4 for future work to develop a cloud storage standard and bridge this gap for data outsourcing. This section is based on our experience from this project, and future work should test our hypothesized generalizations.

6.1 How Mega's Cryptographic Design Fails

We summarize the root causes of our attacks on Mega's architecture (presented in Chapter 4) and determine which attack vectors could apply to other systems.

Key Integrity Protection and RSA-CRT. Our `RSA` key and `AES-ECB` plaintext recovery attacks (Sections 4.1 and 4.2) exploit the missing integrity protection of outsourced key ciphertexts. The client's operation on corrupted key material provides side-channel information that enables us to factor the `RSA` modulus or leak `AES-ECB` plaintexts. Our results show that `RSA-CRT` is particularly vulnerable to attacks in the chosen-plaintext setting since the `RSA-CRT` decryption directly uses the prime factors of the `RSA` modulus, making it more susceptible to leaking information useful for factorizing the modulus.

We note that key outsourcing and RSA-CRT are common design choices. The upload of key ciphertexts enables seamless support of multiple devices: clients use the password to encrypt key material before uploading it to the service provider. Users can simply enter their password on a new device, and their client authenticates them and subsequently fetches and decrypts the key material. Moreover, RSA-CRT is very widely deployed due to reducing computational costs by a factor of four compared to textbook RSA decryption.

Our attacks show the importance of authenticated encryption to protect such outsourced key material. Practitioners might gain the impression that security games for authenticated encryption use too strong adversaries with seemingly unrealistic access to decryption oracles. However, we have shown that (partial) decryption oracles exist in practice, especially in the setting of an untrusted service provider. Moreover, the ability to forge ciphertexts is not only of theoretical concern but can lead to much stronger attacks such as key recovery, as shown in Section 4.1.

Key Separation. Our attacks exploit multiple instances of key reuse: for one, the AES-ECB plaintext recovery attack (Section 4.2) exploits that Mega protects all key material with the same master key and algorithm. Therefore, we can use a vulnerability in the RSA key decryption to compromise unrelated key material. Furthermore, a padding oracle in the legacy chat key decryption enables our variant of Bleichenbacher's attack (Section 4.4) to decrypt shared node keys because both unrelated functionalities use the same public RSA key. Additionally, the extensive key reuse to encrypt files, produce chunk MACs, and compute the metamac leads to many interactions because all instances use the AES block cipher. Although we could not exploit the last key reuse due to Mega's specific structure and the encrypted obfuscated file key, we consider it not unlikely that further attacks are possible. The principle of key separation is a widely accepted best practice in the cryptographic community. Although the interactions between different components are an immediate problem for proofs, concrete failures, such as the ones in Mega, are more intricate and numbered.

Non-Standard Primitives. We found many non-standard uses of cryptographic primitives in Mega's design:

- The key encryption uses the IND-CPA-insecure AES-ECB, which not only lacks integrity protection but is also deterministic. Our key overwriting attacks from Section 4.1 are possible because Mega encrypts keys which are longer than a single AES block with AES-ECB.
- Mega uses CBC-MAC for variable-length concatenations of chunk MACs instead of CMAC.

- Mega's version of AES-CCM deviates from RFC 3610 [164] by not encrypting the MAC tag, which degenerates this mode of operation from a provably secure MAC-then-Encrypt scheme to a not analyzed Encrypt-and-MAC composition.
- Mega uses a custom RSA padding instead of OAEP, which is still vulnerable to a (non-trivial) variant of Bleichenbacher's attack (see Section 4.4).
- Further non-standard uses that we did not discuss previously include legacy code that uses AES-ECB to derive a key from a password and the peculiar structure of the obfuscated file key, which allows an adversary knowing a single plaintext-ciphertext pair to force the file key to be all zero bytes.

These examples reinforce that it is dangerous to "roll your own crypto." Even though some attacks might not be directly applicable, they can often be adapted. Furthermore, some non-standard usages can introduce new vulnerabilities and unintended interactions.

Conclusion. Missing key integrity protection and violating the key separation principle can have severe consequences and lead to concrete attacks. Furthermore, RSA-CRT in a chosen-plaintext setting is susceptible to leak information about factors of the RSA modulus. Moreover, non-standard primitives frequently occur in practice due to custom solutions or implementation mistakes. The resulting algorithms lack the rigorous analysis of proven and standardized primitives and are often vulnerable to (variants of) well-known attacks from the cryptographic literature.

6.2 The Consequences of Mega's Flawed Design

Based on our learnings from the impact of fundamental issues in Mega's cryptographic design, we discuss the severe consequences of a flawed design deployed in a large-scale architecture. In addition to the evident security implications, we conclude from our discussion of countermeasures in Chapter 5 that changing vulnerable architecture poses significant challenges in practice due to backward compatibility and the massive scale. Furthermore, we discuss what security guarantees a customer can hope to obtain after a severe compromise.

Security Implications. Flaws in the cryptographic design of popular applications affect many customers. Such services – especially when they advertise secure encryption and privacy – are an attractive target for resourceful adversaries. For instance, a nation-state actor might be motivated to compromise this service because the probability is high that a targeted person is among

the customers. Moreover, for organized crime, the possible financial gain increases with the number of victims. In both examples, the cost of developing more complex attacks is amortized by the large number of accounts to which it can be applied. Hence, operating popular instantiations of cryptography increases the responsibility to provide a solid design. More resourceful adversaries lead to a higher risk of exploitation and require protection against more complex attacks.

Backward Compatibility. Patching a flaw in the cryptographic design often requires fundamental changes to the architecture. Such disruptive updates are not backward compatible, which causes two main issues: first, users need to update their devices, which is often challenging in practice as different platforms have their own processes and allow more or less control for the developer to enforce updates. Second, simultaneously supporting multiple client versions can enable downgrade attacks on users who have already updated their devices. Moreover, the interaction between different versions, e.g., a new web client and an outdated mobile app of the same account, adds further complexity. Therefore, new systems designs should already prepare for future updates. For instance, a modular design approach with version numbers and scheduled expiration dates facilitates exchanging modules due to later design updates and avoids the existence of severely outdated clients.

Scale. As we discussed in the mitigation in Chapter 5, it can be very challenging to transition large-scale applications to a new architecture. Since Mega stores over 1000 petabytes of data, even with 1000 Gbit/s bandwidth, a re-encryption of all data requires more than half a year. In addition to the immense load on the service provider's infrastructure, all customers need to download, decrypt, encrypt, and upload their data due to end-to-end encryption. However, some clients may not be reachable, or overwhelmed by the task of re-encrypting hundreds of gigabytes of data. Consequently, even if suitable mitigations for a vulnerability are known, it might remain challenging if not infeasible to deploy them. This realization can foster short-term solutions that prevent specific attacks instead of addressing the root causes. Such temporary solutions run into the danger of being insufficient against improved attacks.

Post-Compromise Security. After a severe vulnerability has been detected, customers need in-depth knowledge of the incident to assess what security guarantees the service still provides them after mitigation. Even if a patch is feasible, a severe attack could have compromised all previously stored data and key material. Consequently, re-encryption only restores confidentiality guarantees if the user assumes that their file keys might have been compromised but not yet used to decrypt their data. However, changing

the key material requires trusting the service provider to delete the old and potentially vulnerable ciphertext. Moreover, systems that do not support key rotation might not even allow users to change key material. For instance, Mega's design does not change the share, signature, or chat keys when the user changes or resets the password. In such a case, we cannot achieve post-compromise security: any adversary who has already recovered unchangeable key material or does so based on vulnerable legacy key ciphertexts can break the confidentiality or integrity of data exchanged in the future. Therefore, a defensive implementation should include measures to recover from a severe incident.

Conclusion. We conclude that new system designs should be reviewed carefully by cryptographic experts before their deployment for two reasons: first, widely deployed systems attract resourceful adversaries. Second, mitigating security issues might be infeasible after the release, even if the vulnerability is well understood and a secure solution exists in theory. Furthermore, the architecture should facilitate future modifications and incorporate an infrastructure to deploy updates and prevent severely outdated clients. Finally, it is advisable to prepare a strategy for recovering from possible future compromises.

6.3 Why Mega's Cryptographic Design Fails

In this section, we hypothesize why there is a gap between the cryptographic community and practitioners. In our opinion, the case of Mega shows that, on the one hand, it is challenging to combine existing primitives to a secure system in practice. On the other hand, theoreticians underestimate the practical challenges present in large-scale deployments of cryptographic systems.

Fragile Combinations. Our analysis of Mega shows that well-analyzed components are not enough to build secure end-to-end systems. For instance, using `AES-ECB` to encrypt a single block can be acceptable. However, the missing integrity protection becomes a severe issue when multiple blocks are encrypted because partial key corruption can compromise the preserved secret information. Moreover, neglecting the key separation principle might enable an insecure legacy component to compromise the data of another part of the system.

This fragility of combinations of cryptographic algorithms is not a new observation and has received significant attention in research. New paradigms called universally composable (UC) security [96] and constructive cryptography [137] create frameworks where compositions of building blocks remain secure. However, our analysis of Mega showed that fragile compositions

remain an acute issue in practice. When building systems, developers need to rely on best practices and recommendations in standards. However, it does not suffice to use secure cryptographic primitives due to intricate interactions. Therefore, developers need support from experts to assess the security of their design in the presence of side-channel leakage.

Practical Constraints. We hypothesize that another reason for unusual constructions are practical constraints such as backward compatibility and performance. The optimal solution might not be economically sensible due to high migration costs or potential loss of customers. Furthermore, some backward compatibility can be stipulated (e.g., a contractual agreement to support some generation of mobile phones for at least four years) and force the simultaneous support for new and old architectures enabling downgrade attacks. Moreover, customers highly value usability, and they expect features such as multi-device support and password recovery, which complicate the design. Finally, some legacy code might continue to exist due to the lack of human resources for refactoring. In summary, the non-cryptographic dimension of companies can provide compelling arguments to the executive board to tolerate vulnerable designs in practice if adequate mitigations are costly. Therefore, a system design should facilitate fundamental changes and make mitigations feasible.

Conclusion. Cryptographic designs fail in practice because developers are overwhelmed by the delicate task of combining secure primitives to a solid cryptographic system without expert knowledge. Furthermore, research insufficiently addresses non-security-related constraints leading to knowingly insecure designs persisting in practice. In Section 6.4, we propose to consolidate industry and academic research by developing a standard in future work.

6.4 Future Work: Cloud Storage Standard

It is difficult for practitioners to build a secure system design that provides competitive features for the reasons discussed above. Moreover, many existing cloud providers neither specify their architecture in detail nor provide source code (see Table 3.1). Although providers claim secure end-to-end encryption, which does not require trust in the operator, customers cannot verify these statements. Paradoxically, users need to trust the untrusted cloud provider to design and implement a secure system.

We believe that a secure cloud storage standard can significantly increase the security of cloud solutions by specifying the combination of cryptographic primitives to achieve strong security guarantees and competitive features. We envision an extensible specification that developers can follow and customize

to implement their cloud service. For instance, the standard should support deriving keys for additional features, such as built-in messaging or video conferencing. Furthermore, a modular design should allow services to achieve stronger security notions, including perfect forward secrecy and post-compromise security. The standardized design should account for common practical concerns and provide mechanisms or guidelines to address them. Such a standard avoids that developers need to have expert knowledge. It enables users to have confidence in a cloud service because it implements a public, well-analyzed, and possibly proven standard.

6.5 Conclusion

We surveyed fourteen cloud providers and noticed that they are predominantly closed source and do not document their cryptographic design precisely. Our analysis of Mega – the most popular documented and actively developed cloud storage with over 243 million users – resulted in four severe attacks allowing a malicious service provider or TLS-MitM to break the confidentiality and integrity of user keys and files. Our discussion of countermeasures shows significant practical challenges to address fundamental flaws in widely deployed cryptography due to backward compatibility and the immense strain on the infrastructure for re-encrypting files. Future work can address these issues by designing a practice-oriented cloud storage standard that creates a secure and modular system that is well analyzed and achieves post-compromise security.

Bibliography

[1] A Look Behind the Encryption Scenes – 10th Aniversary Infographics. https://static.boxcryptor.com/infographics/Infographics_10th-Anniversary_EN.png. Retrieved 2021-10-06.

[2] About – Nextcloud. https://nextcloud.com/about/. Retrieved 2021-10-07.

[3] About Us – Encrypted Cloud Storage – MEGA. https://mega.io/about. live user count, retrieved 2022-01-07.

[4] Acceptable Use Policy — Tresorit. https://tresorit.com/acceptable-use-policy. Retrieved 2021-10-21.

[5] asmcrypto.js/aes.asm.js at 4c508d8ffaedf5055d4c6e134530141e1ff8fe9e. https://github.com/asmcrypto/asmcrypto.js/blob/4c508d8ffaedf5055d4c6e134530141e1ff8fe9e/src/aes/aes.asm.js. Retrieved 2021-10-26.

[6] BC File Decryptor. https://github.com/secomba/boxcryptor-single-file-decryptor/. Retrieved 2021-10-06.

[7] BoxCryptor Terms of Use. https://www.boxcryptor.com/en/terms-of-use/. Retrieved 2021-10-21.

[8] Coders' Rights Project Reverse Engineering FAQ, Electronic Frontier Foundation. https://www.eff.org/issues/coders/reverse-engineering-faq. Retrieved 2022-01-13.

[9] Crypto++ Wiki. https://www.cryptopp.com/wiki/Main_Page. Retrieved 2021-10-25.

[10] crypto.getRandomValues() — Can I use... Support tables for HTML5, CSS3, etc. https://caniuse.com/getrandomvalues. Retrieved 2021-10-26.

[11] Cryptopp: Security Vulnerabilities. https://www.cvedetails.com/vulnerability-list.php?vendor_id=15519&product_id=0&version_id=0&page=1&hasexp=0&opdos=0&opec=0&opov=0&opcsrf=0&opgpriv=0&opsqli=0&opxss=0&opdirt=0&opmemc=0&ophttprs=0&opbyp=0&opfileinc=0&opginf=0&cvssscoremin=0&cvssscoremax=0&year=0&cweid=0&order=1&trc=7&sha=0eec8b6b922ea9be895311786a2d1b386f05a9ce. Retrieved 2021-10-25.

[12] Download Site terms — Tresorit. https://tresorit.com/terms-of-use. Retrieved 2021-10-21.

[13] Dropbox – Terms of Service. https://www.dropbox.com/terms. Retrieved 2021-10-21.

[14] Dropbox: Cloud Storage & Drive — androidrank.org. https://www.androidrank.org/application/dropbox_cloud_storage_drive/com.dropbox.android. Retrieved 2021-10-07.

[15] Dropbox Open Source. https://opensource.dropbox.com/. Retrieved 2021-10-06.

[16] Dropbox Quarterly Filings, Form 10-Q (2021, August 6). https://investors.dropbox.com/financial-information/sec-filings. Retrieved 2021-10-06.

[17] GDPR – Nextcloud Administration Manual. https://docs.nextcloud.com/server/21/admin_manual/gdpr/index.html. Retrieved 2021-10-28.

[18] GitHub meganz/webclient: strongvelope.js legacy RSA chat key decryption. https://github.com/meganz/webclient/blob/8b8126f1fbf2ee1d82831bdf4b3e77f21bcbeb24/js/chat/strongvelope.js#L889. Retrieved 2022-01-06.

[19] Google Drive — androidrank.org. https://www.androidrank.org/application/google_drive/com.google.android.apps.docs. Retrieved 2021-10-07.

[20] Google Drive Terms of Service. https://www.google.com/drive/terms-of-service/. Retrieved 2021-10-21.

[21] Google Open Source – opensource.google. https://opensource.google/?from=CloudNativeSummit. Retrieved 2021-10-07.

[22] Google Terms of Service – Privacy & Terms – Google. https://policies.google.com/terms. Retrieved 2021-10-21.

[23] How Reliable is MEGA's End-to-End Encrypted Storage? – MEGA. https://mega.nz/about/reliability. Retrieved 2022-02-01.

[24] Icedrive – Free Cloud Storage — androidrank.org. https://www.androidrank.org/application/icedrive_free_cloud_storage/com.icedrive.app. Retrieved 2021-10-07.

[25] Icedrive – Secure Encrypted Cloud Storage. https://icedrive.net/encrypted-cloud-storage. Retrieved 2021-10-07.

[26] Interviews: Kim Dotcom Answers Your Questions. https://yro.slashdot.org/story/15/07/27/200204/interviews-kim-dotcom-answers-your-questions?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+Slashdot%2Fslashdot+%28Slashdot%29. Retrieved 2022-01-10.

[27] Keybase. https://keybase.io. Retrieved 2022-02-23.

[28] Keybase Client Repo. https://github.com/keybase/client. Retrieved 2022-02-23.

[29] License - Seafile Admin Manual. https://manual.seafile.com/deploy_pro/seafile_professional_sdition_software_license_agreement/. Retrieved 2021-10-21.

[30] MEGA — androidrank.org. https://www.androidrank.org/application/mega/mega.privacy.android.app. Retrieved 2021-10-07.

[31] MEGA LIMITED (4136598) Registered. https://app.companiesoffice.govt.nz/companies/app/ui/pages/companies/4136598/shareholdings. Retrieved 2022-01-10.

[32] Nextcloud developer documentation. https://docs.nextcloud.com/server/latest/developer_manual/. Retrieved 2021-10-07.

[33] Nextcloud FAQ: How to share End-to-End encrypted folder with other users? https://help.nextcloud.com/t/how-to-share-end-to-end-encrypted-folder-with-other-users/98696/12/. Retrieved 2021-10-20.

[34] Nextcloud GitHub Issue #2490. https://github.com/nextcloud/desktop/issues/2490/. Retrieved 2021-10-20.

[35] Nextcloud GitHub Repository. https://github.com/nextcloud. Retrieved 2021-10-07.

[36] openssl/aes_x86core.c at master. https://github.com/openssl/openssl/blob/master/crypto/aes/aes_x86core.c. Retrieved 2021-10-26.

[37] pCloud – Developers. https://docs.pcloud.com/. Retrieved 2021-10-07.

[38] pCloud – Europe's Most Secure Cloud Storage. https://www.pcloud.com/eu. Retrieved 2021-10-07.

[39] pCloud – Terms and Conditions. https://www.pcloud.com/terms_and_conditions.html. Retrieved 2021-10-21.

[40] pCloud features – Encryption. https://www.pcloud.com/features/crypto.html. Retrieved 2021-10-07.

[41] pCloud GitHub. https://github.com/pcloudcom. Retrieved 2021-10-21.

[42] RNGCryptoServiceProvider Class — Microsoft Docs. https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.rngcryptoserviceprovider?view=net-5.0. Retrieved 2021-10-29.

[43] Seafile – Open Source File Sync and Share Software. https://www.seafile.com/en/home/. Retrieved 2021-10-08.

[44] Seafile Server GitHub Repository. https://github.com/haiwen/seafile-server. Retrieved 2021-10-08.

[45] Seafile Sync Client Daemon GitHub Repository. https://github.com/haiwen/seafile. Retrieved 2021-10-08.

[46] Security features - Seafile Admin Manual. https://manual.seafile.com/security/security_features/. Retrieved 2021-10-08.

[47] Security Principles – Syncthing v1 documentation. https://docs.syncthing.net/users/security.html/. Retrieved 2021-10-08.

[48] Software Developer Documentation – MEGA. https://mega.nz/doc. Retrieved 2021-10-29.

[49] SonarQube: Code Quality and Code Security. https://www.sonarqube.org/. Retrieved 2021-10-28.

[50] Syncthing GitHub Repository. https://github.com/syncthing/syncthing. Retrieved 2021-10-08.

[51] Syncthing Usage Reports. https://data.syncthing.net/. Retrieved 2021-10-08.

[52] Terms of Service — Sync. https://www.sync.com/terms. Retrieved 2021-10-21.

[53] The GDPR – pCloud's road to full compliance. https://www.pcloud.com/gdpr/. Retrieved 2021-10-07.

[54] Third Party sources. https://tresorit.com/third-party-code. Retrieved 2021-10-08.

[55] Transparency and Public Source Code – MEGA. https://mega.io/sourcecode. Retrieved 2021-10-06.

[56] File Security in Microsoft SharePoint and OneDrive for Business. https://www.microsoft.com/en-us/download/details.aspx?id=53884&culture=en-us&country=US, September 2016. Retrieved 2021-10-07.

[57] ownCloud Statement concerning the formation of Nextcloud by Frank Karlitschek. https://owncloud.com/news/owncloud-statement-concerning-formation-nextcloud-frank-karlitschek/, June 2016. Retrieved 2021-10-27.

[58] Worried about Kim Dotcom's tweet concerning MEGA? https://mega.io/blog/worried-about-kim-dotcoms-tweet-concerning-mega, April 2016. Retrieved 2022-01-10.

[59] Encryption in Transit in Google Cloud. https://cloud.google.com/security/encryption-in-transit/, December 2017. Retrieved 2021-10-07.

[60] Google Cloud Security Whitepapers. https://services.google.com/fh/files/misc/security_whitepapers_march2018.pdf, March 2018. Retrieved 2021-10-07.

[61] Google security whitepaper. https://cloud.google.com/security/overview/whitepaper, January 2019. Retrieved 2021-10-07.

[62] MEGA Security White Paper. https://mega.nz/SecurityWhitepaper.pdf, January 2020. Retrieved 2021-10-06.

[63] Crypto.getRandomValues() – Web APIs — MDN. https://developer.mozilla.org/en-US/docs/Web/API/Crypto/getRandomValues, October 2021. Retrieved 2021-10-26.

[64] Dropbox Business Security Whitepaper, version V2021.06. https://assets.dropbox.com//www/en-us/business/solutions/solutions/dfb_security_whitepaper.pdf, 2021. Retrieved 2021-10-06.

[65] Eight years of MEGA – Tweet. https://twitter.com/MEGAprivacy/status/1352564229044277248?s=20, January 2021. Retrieved 2022-01-10.

[66] iCloud Security Overview – Apple Support. https://support.apple.com/en-us/HT202303, October 2021. Retrieved 2022-02-23.

[67] Legal – iCloud – Apple. https://www.apple.com/legal/internet-services/icloud/en/terms.html, September 2021. Retrieved 2022-02-24.

[68] Mega Transparency Report. https://mega.io/Mega_Transparency_Report_September_2021.pdf, September 2021. Retrieved 2021-10-21.

[69] Microsoft Services Agreement. https://www.microsoft.com/en-us/servicesagreement, April 2021. Retrieved 2021-10-21.

[70] Privacy and Data Policy – GDPR – MEGA. https://mega.io/privacy, January 2021. Retrieved 2021-10-25.

[71] Terms of Service – MEGA. https://mega.io/terms, January 2021. Retrieved 2021-10-21.

[72] Data Encryption in OneDrive for Business and SharePoint Online — Microsoft Docs. https://docs.microsoft.com/en-us/microsoft-365/compliance/data-encryption-in-odb-and-spo?view=o365-worldwide, 2021-09-20. Retrieved 2021-10-07.

[73] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. RACS: a case for cloud storage diversity. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 229–240, 2010.

[74] Onur Acıiçmez and Çetin Kaya Koç. Trace-driven cache attacks on AES (short paper). In *International Conference on Information and Communications Security*, pages 112–121. Springer, 2006.

[75] Onur Acıiçmez, Werner Schindler, and Çetin K Koç. Cache based remote timing attack on the AES. In *Cryptographers' track at the RSA conference*, pages 271–286. Springer, 2007.

[76] Nadhem J Al Fardan and Kenneth G Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540. IEEE, 2013.

[77] Ange Albertini, Thai Duong, Shay Gueron, Stefan Kölbl, Atul Luykx, and Sophie Schmieg. How to Abuse and Fix Authenticated Encryption Without Key Commitment. *IACR Cryptol. ePrint Arch.*, 2020:1456, 2020.

[78] Joseph Amankwah-Amoah, Zaheer Khan, Geoffrey Wood, and Gary Knight. Covid-19 and digitalization: The great acceleration. *Journal of Business Research*, 136:602–611, 2021.

[79] Scott Arciszewski. Preventing timing attacks on string comparison with a double hmac strategy – paragon initiative enterprises blog. https://paragonie.com/blog/2015/11/preventing-timing-attacks-on-string-comparison-with-double-hmac-strategy, November 2015. Retrieved 2021-10-25.

[80] C Ashokkumar, Ravi Prakash Giri, and Bernard Menezes. Highly efficient algorithms for AES key retrieval in cache access attacks. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 261–275. IEEE, 2016.

[81] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, and Joe-Kai Tsay. Efficient Padding Oracle Attacks on Cryptographic Hardware. In *Annual Cryptology Conference*, pages 608–625. Springer, 2012.

[82] Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 92–111. Springer, 1994.

[83] Daniel J. Bernstein. Cache-timing attacks on AES. 2005.

[84] Daniel J Bernstein. Curve25519: new Diffie-Hellman speed records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer, 2006.

[85] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of cryptographic engineering*, 2(2):77–89, 2012.

[86] Guido Bertoni, Vittorio Zaccaria, Luca Breveglieri, Matteo Monchiero, and Gianluca Palermo. AES power attack based on induced cache miss and countermeasure. In *International Conference on Information Technology: Coding and Computing (ITCC'05)-Volume II*, volume 1, pages 586–591. IEEE, 2005.

[87] John Black and Phillip Rogaway. CBC MACs for arbitrary-length messages: The three-key constructions. *Journal of cryptology*, 18(2):111–131, 2005.

[88] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS# 1. In *Annual International Cryptology Conference*, pages 1–12. Springer, 1998.

[89] Enrico Bocchi, Idilio Drago, and Marco Mellia. Personal cloud storage benchmarks and comparison. *IEEE Transactions on Cloud Computing*, 5(4):751–764, 2015.

[90] Hanno Böck, Juraj Somorovsky, and Craig Young. Return Of Bleichenbacher's Oracle Threat (ROBOT). In *27th USENIX Security Symposium (USENIX Security 18)*, pages 817–849, 2018.

[91] Andrey Bogdanov, Thomas Eisenbarth, Christof Paar, and Malte Wienecke. Differential cache-collision timing attacks on AES with applications to embedded CPUs. In *Cryptographers' Track at the RSA Conference*, pages 235–251. Springer, 2010.

[92] Alexandra Boldyreva. Strengthening security of RSA-OAEP. In *Cryptographers' Track at the RSA Conference*, pages 399–413. Springer, 2009.

[93] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *International*

conference on the theory and applications of cryptographic techniques, pages 37–51. Springer, 1997.

[94] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 201–215. Springer, 2006.

[95] Thomas Brewster. Kim Dotcom's Mega Fileshare Service Riddled With Security Holes. https://www.silicon.co.uk/workspace/kim-dotcom-mega-fileshare-security-law-105024, January 2013. Retrieved 2021-10-21.

[96] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[97] CFRG. Results of the PAKE selection process. https://datatracker.ietf.org/meeting/interim-2020-cfrg-01/materials/slides-interim-2020-cfrg-01-sessa-results-of-the-pake-selection-process-00, April 2020.

[98] Gabrielle De Micheli and Nadia Heninger. Recovering cryptographic keys from partial information, by example. 2020.

[99] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. Fast Message Franking: From Invisible Salamanders to Encryptment. In *Annual International Cryptology Conference*, pages 155–186. Springer, 2018.

[100] Markus Dürmuth, Tim Güneysu, Markus Kasper, Christof Paar, Tolga Yalcin, and Ralf Zimmermann. Evaluation of standardized password-based key derivation against parallel processing platforms. In *European Symposium on Research in Computer Security*, pages 716–733. Springer, 2012.

[101] Morris J Dworkin. SP 800-38D. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, 2007.

[102] Morris J Dworkin et al. NIST SP 800-38B. Recommendation for block cipher modes of operation: The CMAC mode for authentication. 2016.

[103] William F Ehrsam, Carl HW Meyer, John L Smith, and Walter L Tuchman. Message verification and transmission error detection by block chaining, February 1978. US Patent 4,074,066.

[104] Niels Ferguson. Authentication weaknesses in GCM. Comments submitted to NIST Modes of Operation Process, 2005.

[105] Dawn Foster. NextCloud Revives ownCloud's Open Source Cloud Storage Software – The New Stack. https://thenewstack.io/story-behind-nextcloud/, July 2016. Retrieved 2021-10-27.

[106] David Freedman and Persi Diaconis. On the histogram as a density estimator: L 2 theory. *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete*, 57(4):453–476, 1981.

[107] Harvey L Garner. The Residue Number System. In *Papers presented at the the March 3-5, 1959, western joint computer conference*, pages 146–153, 1959.

[108] Danilo Gligoroski, Suzana Andova, and Svein Johan Knapskog. On the importance of the key separation principle for different modes of operation. In *International Conference on Information Security Practice and Experience*, pages 404–418. Springer, 2008.

[109] Matthew Green. Looking back at the Snowden revelations. https://blog.cryptographyengineering.com/2019/09/24/looking-back-at-the-snowden-revelations/, September 2019.

[110] Shay Gueron and Vlad Krasnov. The fragility of aes-gcm authentication algorithm. Cryptology ePrint Archive, Report 2013/157, 2013. https://ia.cr/2013/157.

[111] D. Gullasch, E. Bangerter, and S. Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505, May 2011.

[112] Berk Gülmezoğlu, Mehmet Sinan Inci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. A faster and more realistic flush+ reload attack on AES. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 111–126. Springer, 2015.

[113] Nicholas A Howgrave-Graham. *Computational mathematics inspired by RSA*. University of Bath (United Kingdom), 1998.

[114] Bonan Huang. Cache-collision timing attacks against AES-GCM, 2010.

[115] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a Minute! A fast, Cross-VM Attack on AES. pages 299–319, 09 2014.

[116] ITU. X.509: Information technology - Open Systems Interconnection - The Directory: Public-key and attribute certificate frameworks. https://www.itu.int/rec/T-REC-X.509. Retrieved 2021-10-27.

[117] Tibor Jager, Sebastian Schinzel, and Juraj Somorovsky. Bleichenbacher's attack strikes again: breaking PKCS# 1 v1. 5 in XML Encryption. In *European Symposium on Research in Computer Security*, pages 752–769. Springer, 2012.

[118] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 456–486. Springer, 2018.

[119] Adrianne Jeffries. Pirates beware: Kim Dotcom's Mega isn't the safe haven he says it is — The Verge. https://www.theverge.com/2013/1/31/3933774/kim-dotcoms-new-site-mega-is-a-flop-with-pirates-and-heres-why, January 2013. Retrieved 2021-10-21.

[120] Jakob Jonsson and Burt Kaliski. Public-key cryptography standards (PKCS)# 1: RSA cryptography specifications version 2.1. Technical report, RFC 3447, February, 2003.

[121] Antoine Joux. Authentication failures in NIST version of GCM. *NIST Comment*, page 3, 2006.

[122] Seny Kamara and Kristin Lauter. Cryptographic cloud storage. In *International Conference on Financial Cryptography and Data Security*, pages 136–149. Springer, 2010.

[123] Frank Karlitschek. big changes: I am leaving ownCloud, Inc. today. https://karlitschek.de/2016/04/big-changes-i-am-leaving-owncloud-inc-today/, April 2016. Retrieved 2021-10-27.

[124] Emilia Käsper and Peter Schwabe. Faster and Timing-Attack Resistant AES-GCM. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 1–17. Springer, 2009.

[125] Gregg Keizer. Microsoft's OneDrive changes: Follow the money. https://www.computerworld.com/article/3003140/microsofts-onedrive-changes-follow-the-money.html, 2015-11-09. Retrieved 2021-10-07.

[126] keybase. Keybase Book: Keybase Crypto Documents. https://book.keybase.io/docs/crypto. Version 2.0, Retrieved 2022-02-23.

[127] Vlastimil Klíma and Tomas Rosa. Attack on Private Signature Keys of the OpenPGP Format, PGP (TM) Programs and Other Applications Compatible with OpenPGP. *IACR Cryptol. ePrint Arch.*, 2002:76, 2002.

[128] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *Annual Cryptology Conference*, pages 631–648. Springer, 2010.

[129] Hugo Krawczyk and Pasi Eronen. HMAC-based extract-and-expand key derivation function (HKDF). Technical report, RFC 5869, May, 2010.

[130] Burak Kulaksizoglu. GitHub Issue #24577: KeyBase is DEAD. https://github.com/keybase/client/issues/24577, July 2021.

[131] RSA Laboratories. PKCS #5: Password-Based Encryption Standard Version 2.1, October 2012.

[132] Frederic Lardinois. Google Drive will hit a billion users this week — TechCrunch . https://techcrunch.com/2018/07/25/google-drive-will-hit-a-billion-users-this-week/, 2018-07-25. Retrieved 2021-10-07.

[133] Nate Lawson. Side-channel attacks on cryptographic software. *IEEE Security & Privacy*, 7(6):65–68, 2009.

[134] Julia Len, Paul Grubbs, and Thomas Ristenpart. Partitioning Oracle Attacks. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 195–212. USENIX Association, 2021.

[135] Arjen K Lenstra. Memo on RSA signature generation in the presence of faults. Technical report, 1996.

[136] Arjen K Lenstra, Hendrik Willem Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische annalen*, 261(ARTICLE):515–534, 1982.

[137] Ueli Maurer. Constructive cryptography–a new paradigm for security definitions and proofs. In *Joint Workshop on Theory of Security and Applications*, pages 33–56. Springer, 2011.

[138] Alexander May. *Using LLL-Reduction for Solving RSA and Factorization Problems*, pages 315–348. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[139] Robert Mazzoli. SharePoint and OneDrive data resiliency in Microsoft 365 — Microsoft Docs. https://docs.microsoft.com/en-us/compliance/assurance/assurance-sharepoint-onedrive-data-resiliency, 2021-08-27. Retrieved 2021-10-07.

[140] David McGrew and John Viega. The Galois/counter mode of operation (GCM). *submission to NIST Modes of Operation Process*, 20:0278–0070, 2004.

[141] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. Revisiting SSL/TLS implementations: New bleichenbacher side channels and attacks. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 733–748, 2014.

[142] Daniele Micciancio and Shafi Goldwasser. Shortest vector problem. In *Complexity of Lattice Problems*, pages 69–90. Springer, 2002.

[143] Steve Morgan. 2020 Data Attack Surface Report. Cybersecurity Ventures, 12 2020.

[144] Kathleen Moriarty, Burt Kaliski, and Andreas Rusch. PKCS# 5: Password-based cryptography specification version 2.1. *Internet Eng. Task Force (IETF)*, 8018:1–40, 2017.

[145] Maurizio Naldi and Loretta Mastroeni. Cloud storage pricing: A comparison of current practices. In *Proceedings of the 2013 international workshop on Hot topics in cloud services*, pages 27–34, 2013.

[146] Nextcloud. End-to-End Encryption Design. https://nextcloud.com/whitepapers/, 2017-09-20.

[147] Nextcloud. Server-side Encryption. https://nextcloud.com/whitepapers/, June 2018.

[148] Jordan Novet. The case for Apple to sell a version of iCloud for work. https://www.cnbc.com/2018/02/11/apple-could-sell-icloud-for-the-enterprise-barclays-says.html, February 2018. Retrieved 2022-02-23.

[149] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. January 2005.

[150] Jos Poortvliet. Nextcloud grew customer base 7x, added over 6.6 million lines of code and doubled its team in 2017. https://nextcloud.com/blog/nextcloud-grew-customer-base-7x-added-over-6.6-million-lines-of-code-and-doubled-its-team-in-2017/, 2018-01-11. Retrieved 2021-10-07.

[151] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[152] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. The 9 Lives of Bleichenbacher's CAT: New Cache ATtacks on TLS Implementations. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 435–452. IEEE, 2019.

[153] Secomba GmbH. Technical Overview – How Boxcryptor's Encryption Works. https://www.boxcryptor.com/en/technical-overview/. Retrieved 2021-10-06.

[154] Emily Stark, Michael Hamburg, and Dan Boneh. Symmetric Cryptography in JavaScript. In *2009 Annual Computer Security Applications Conference*, pages 373–381. IEEE, 2009.

[155] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. In *Annual International Cryptology Conference*, pages 570–596. Springer, 2017.

[156] sync.com. Privacy White Paper. https://www.sync.com/pdf/sync-privacy-whitepaper.pdf, September 2015. Retrieved 2021-10-08.

[157] Tresorit Team. Cryptography and cybersecurity veteran David Canellos joins Tresorit's board to fuel worldwide growth. https://tresorit.com/blog/cryptography-and-cybersecurity-veteran-david-canellos-joins-tresorits-board-to-fuel-worldwide-growth/, 2019-02-28. Retrieved 2021-10-08.

[158] tresorit. Tresorit Encryption Whitepaper. https://tresorit.com/resources, https://web.tresorit.com/l/fyvwX#QTQCXngLXUIzWC0hghSJoQ. Retrieved 2021-10-08.

[159] tresorit. Tresorit Whitepaper. https://tresorit.com/files/tresoritwhitepaper.pdf. Retrieved 2021-10-08.

[160] Nikos Virvilis, Stelios Dritsas, and Dimitris Gritzalis. Secure cloud storage: Available infrastructures and architectures review and evaluation. In *International Conference on Trust, Privacy and Security in Digital Business*, pages 74–85. Springer, 2011.

[161] Alex von Gluck. GitHub Issue #24105: Open source the server components of Keybase. https://github.com/keybase/client/issues/24105, May 2020. Retrieved 2022-02-23.

[162] Michael Weiß, Benedikt Heinz, and Frederic Stumpf. A cache timing attack on AES in virtualization environments. In *International Conference*

on Financial Cryptography and Data Security, pages 314–328. Springer, 2012.

[163] Daniel Lowe Wheeler. zxcvbn: Low-Budget Password Strength Estimation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 157–173, Austin, TX, August 2016. USENIX Association.

[164] Doug Whiting, Russ Housley, and Niels Ferguson. Counter with CBC-MAC (CCM). 2003.

[165] Jiyi Wu, Lingdi Ping, Xiaoping Ge, Ya Wang, and Jianqing Fu. Cloud storage as the infrastructure of cloud computing. In *2010 International Conference on Intelligent Computing and Cognitive Informatics*, pages 380–383. IEEE, 2010.

[166] Xianglong Wu, Rui Jiang, and Bharat Bhargava. On the security of data access control for multiauthority cloud storage systems. *IEEE Transactions on Services Computing*, 10(2):258–272, 2015.

[167] Eric S. Yuan. Zoom Acquires Keybase and Announces Goal of Developing the Most Broadly Used Enterprise End-to-End Encryption Offering. https://blog.zoom.us/zoom-acquires-keybase-and-announces-goal-of-developing-the-most-broadly-used-enterprise-end-to-end-encryption-offering/, May 2020.

[168] Wenying Zeng, Yuelong Zhao, Kairi Ou, and Wei Song. Research on cloud storage architecture and key technologies. In *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, pages 1044–1048, 2009.

[169] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 990–1003, 2014.

Appendix A

Glossary

API Application Programming Interface; a set of functions provided by one computer system to another

CRT Chinese Remainder Theorem; states in a simplified form that the system of equations $m \equiv_p a_1$ and $m \equiv_q a_2$ has a unique solution $m \mod p \cdot q$ for coprime p, q. The theorem has a constructive proof to efficiently obtain this solution.

CSPRNG Cryptographically Secure Pseudo-Random Number Generator; PRNG producing sufficient entropy for cryptographic applications

CVE Common Vulnerabilities and Exposures; a public list of disclosed vulnerabilities, categorized by products and severity

E2EE End-to-End Encryption; a primitive where data is encrypted on the end users' devices and the encryption keys never leave those devices

Encrypt-then-MAC Encrypt-then-MAC; generic composition of an encryption and a message authentication scheme, where we first encrypt the plaintext and then create a MAC tag on the ciphertext

Encrypt-and-MAC Encrypt-and-MAC; generic composition of a message authentication and an encryption scheme, where we concatentate the MAC of the plaintext to the ciphertext

GDPR General Data Protection Regulation; a European regulation governing the data protection and privacy of European citizens

IND-CPA INDistinguishability against a Chosen-Plaintext Attacker; a common security game in cryptography, where an attacker can submit two plaintexts to an encryption oracle and then has to guess which one was encrypted

IV Initialization Vector; a randomly chosen but not confidential set of bits that is used to initialize a cryptographic algorithm

LSB Least Significant Bits; the rightmost bits of an integer; i.e., the k LSBs are $b_{k-1}b_{k-2}\dots b_0$ of an integer $i \leftarrow \sum_{i=0}^{n}(b_i \cdot 2^i)$.

MAC Message Authentication Code; a keyed, symmetric primitive to verify the integrity of data

MAC-then-Encrypt MAC-then-Encrypt; generic composition of a message authentication and an encryption scheme, where we first create the MAC of the plaintext and then encrypt the both the plaintext and the MAC

MitM Man-in-the-Middle or Person-in-the-Middle (PitM); an adversarial strategy, where the malicious entity eavesdrops on the communication and possibly manipulates exchanged messages

MSB Most Significant Bits; the leftmost bits of an integer; i.e., the k MSBs are $b_n b_{n-1}\dots b_{n-k+1}$ of an integer $i \leftarrow \sum_{i=0}^{n}(b_i \cdot 2^i)$.

nonce Number only used once; there are various instanciations (counter, random value) and the security of schemes building on nonces relies on the uniqueness (w.h.p.) of them

PKW Puncturable Key Wrapping; an extension of key wrapping with the ability to puncture the master secret key and thereby for specific, wrapped keys, making them unrecoverable

SDK Software Development Kit; a set of tools, usually exposed as API, that implements software-specific functionality that developers can use to build applications.

SID Session Identifier; a secret token, chosen (often randomly) by the server and sent to the client after successful authentication. The client needs to resend this token with subsequent requests identify itself.

TEE Trusted Execution Environment; an isolated processor area where sensitive code can be executed securely despite not trusting the entire platform

TLS-MitM TLS Man-in-the-Middle; a MitM attack where the adversary has compromised the `TLS` connection (e.g., by installing a malicious root certificate)

TOFU Trust On First Use; a trust assumption that the first connection to a service happens in an uncompromised environment and exchanged information can therefore be trusted and stored

ToS Terms of Service; a legal agreement that users of a service need to accept, also known as Terms of Use

Appendix B

Nextcloud's Cryptographic Design

Nextcloud is an exciting research target because it is an entirely open-source product with a respectably sized user community. Therefore, we can investigate both server and client source code. In addition, they describe their design and aim for E2EE.

Regarding the history of Nextcloud, Frank Karlitschek forked this project in 2016 from ownCloud. He previously started ownCloud in 2010 and later co-founded ownCloud Inc. Karlitschek explains in a blog post [123] to leave ownCloud Inc due to conflicting opinions on the company's long-term focus. He prefers to involve the community more and to prioritize their contribution over financial goals. Many senior staff members left ownCloud along with the founder and joined his new project. Consequently, ownCloud Inc was forced to shut down [57], but ownCloud GmbH still continues to maintain the ownCloud project. Karlitschek's new Nextcloud GmbH does not have an enterprise edition like ownCloud. Instead, their business model focuses on support and consulting [105].

Nextcloud provides two white papers explaining their security modes: server-side encryption [147] and client-side E2EE [146]. A user can selectively enable the latter mode for individual folders. We note that the documentation from 2017 warns that the "end-to-end encryption feature is a work-in-progress and this document may describe functionalities or approaches not yet implemented in our testing releases" [146]. Indeed, we discovered that sharing of E2EE folders is not yet supported.

B.1 Architecture

In the following, we present Nextcloud's design of the registration process (Appendix B.1.1) and client authentication (Appendix B.1.2). Furthermore, we describe their outlines for node encryption (Appendix B.1.3) and shar-

ing features (Appendix B.1.4). Finally, we briefly discuss account recovery (Appendix B.1.5).

B.1.1 Registration

Algorithm 19 describes the two registration cases for a user. For the first device, the Nextcloud client generates a new `RSA` key pair and requests a certificate from the server. Next, the client selects 12 random words from a set of 2048 English words $\mathcal{W}$. This so-called mnemonic is used together with a random salt s to derive a key k. In other words, the mnemonic replaces a password with the limitation that it is only used for enabling client-side encryption on a new device. The client stores the mnemonic in the device's key chain, but the user is also responsible for safely keeping it offline. The client uses k to encrypt the secret key sk with `AES-GCM` and upload the result together with the salt s, nonce n, and tag τ to the server. In the other case, when the user has already registered a device, the client will retrieve the encrypted secret key, salt s, nonce n, and tag τ from the server. It then derives the key k from the salt s and the user-provided mnemonic m and decrypts the secret key sk with k. Before the client stores the key pair (sk, pk) in the local keychain, it verifies that the secret key sk belongs to pk from the client's certificate by encrypting and decrypting 512 random bits. The client fetches the user's certificate from the server and verifies it using the server's public key, which it requests too if it is not already available.

B.1.2 Client Authentication

The white papers only focus on encryption and do not describe authentication. A user has a regular password in addition to the mnemonic. The client only uses the latter to protect the encrypted folders' asymmetric key material. However, not all data is encrypted, and instead of the mnemonic, Nextcloud uses the password to authenticate the user. In the login procedure, the user transmits his password over the `HTTPS` connection in cleartext. Depending on the used backend, the server hashes the password and compares it with a stored value. If these match, the user is authenticated.

Appendix B.2.4 discusses why this authentication does not compromise E2EE despite trusting the server.

B.1.3 Node Encryption

Algorithm 20 describes the procedure of uploading a file or folder F to a shared encrypted folder D. Note that Nextcloud does not yet implement the sharing-related part of this pseudo-code. To upload F, the client first fetches the metadata file of D from the server. Next, it selects a file key k_F uniformly at random and uses it to encrypt F with `AES-GCM`. Afterward, the

Algorithm 19 Nextcloud's procedure to register a client with user ID *uid*

1: **procedure** REGISTER_USER(uid)
2: **if** is first device **then**
▷ Generate keys
3: $sk, pk \xleftarrow{\$} \texttt{RSA.Gen}(2048)$
4: $csr \leftarrow \texttt{X509.CSR}(sk, pk, uid)$
5: $cert \leftarrow \texttt{server.get\_cert}(csr)$

▷ Store keys
6: $m \subseteq_{\$} \mathcal{W}, |m| = 12$ ▷ $\mathcal{W}$ is a set of words, $|\mathcal{W}| = 2048$
7: $\texttt{keychain.Put}((pk, sk), m)$
8: $s \leftarrow_{\$} \{0,1\}^{320}$ ▷ salt
9: $k \leftarrow \texttt{PBKDF2-HMAC-SHA1}(m, s, \text{iter=1024}, \text{len=256})$
10: $n \leftarrow_{\$} \{0,1\}^{128}$
11: $[sk]_k, \tau \leftarrow \texttt{AES-GCM.Enc}(k, sk, n, \emptyset)$ ▷ nonce
12: $\texttt{server.store}([sk]_k, s, n, \tau)$
13: **else**
14: $m \leftarrow \texttt{user\_input}(\text{'Enter mnemonic: '})$
15: $[sk]_k, s, n, \tau \leftarrow \texttt{server.get\_priv\_key}(uid)$
16: $k \leftarrow \texttt{PBKDF2-HMAC-SHA1}(m, s, \text{iter=1024}, \text{len=256})$
17: $sk \leftarrow \texttt{AES-GCM.Dec}(k, n, \tau, [sk]_k, \emptyset)$
18: $cert \leftarrow \texttt{server.get\_cert}(uid)$

▷ Verify secret key
19: $r \leftarrow_{\$} \{0,1\}^{512}$
20: $[r]_{sk} \leftarrow \texttt{RSA-ECB-OAEPWithSHA-256AndMGF1Padding.Enc}(cert.pk, r)$
21: $r' \leftarrow \texttt{RSA-ECB-OAEPWithSHA-256AndMGF1Padding.Dec}(sk, [r]_{sk})$
22: **if** $r' \neq r$ **then**
23: **return** $\perp$
24: $\texttt{keychain.Put}((sk, cert.pk), m)$

client recovers all metadata key versions $k_{md}^{(i)}$ stored in `md.keys` and encrypted with the client's public key pk. The client needs to pick the most recent key $k_{md}^{(i_{max})}$ for encrypting the new file key k_F. All other keys are outdated and only maintained because the file metadata is lazily re-encrypted with newer keys once it is accessed. We have to distinguish two cases for picking the file identifier. If another file g already exists at this path, we reuse g's identifier id_g and thus overwrite the file. Otherwise, we select a new identifier id_f uniformly at random. As the last step, the client updates the metadata. For this, it first obtains a locking token t from the server. Next, it uploads the new file and then modifies the metadata to add the new file information – encrypted with the file metadata key – to `md.files`. Finally, the client releases the `md` lock again. Note that the client uses the locking token t in subsequent server requests to prove lock ownership. Without an implementation, it is unclear how Nextcloud plans to recover when the client holding a token suddenly goes offline. If this is not done carefully, an adversary could deceive a client into reverting another user's changes in the metadata unintentionally.

B.1.4 Node Sharing

Algorithm 22 and Algorithm 23 show the intended design of Nextcloud's sharing feature for E2EE folders. However, a GitHub issue [34] and FAQ entry [33] confirmed together with the source code [35] that Nextcloud has not yet implemented this feature. Consequently, some details of the procedures described in this section might differ from the final implementation.

Every E2EE folder has a `md` file, storing a metadata key k_{md} encrypted for every sharer. This key is used to encrypt all content of the `md` file. Furthermore, it stores $\mathcal{R}$, the set of entities with whom the folder is shared, together with their public keys. Finally, the `md` file holds file metadata – the file path, MIME type, and file encryption key.

Algorithm 21 shows how Nextcloud's client sets up a new E2EE folder: first, it marks the directory as encrypted. Second, it samples a uniformly random metadata key $k_{md}^{(0)}$. The exponent stands for the key version, which is useful for key rotation, e.g., for the purpose of removing access of former members to a shared folder (see Algorithm 23). Next, the client encrypts the new metadata key with the public keys of all recipients using `RSA-ECB-OAEPWithSHA-256AndMGF1Padding` and stores them for key version 0 in `md.keys`. Finally, it stores all recipients and their public keys, encrypted with $k_{md}^{(0)}$ and `AES-GCM`, in the metadata file. In this way, Nextcloud does not disclose the sharers' identities from the metadata file. However, the folder members can decrypt $k_{md}^{(0)}$ and then learn the other sharers.

Algorithm 22 shows how a user r shares a folder D with another user s. Nextcloud employs Trust On First Use (TOFU): if the client has not seen

Algorithm 20 Nextcloud's procedure to upload a new node F to an E2EE folder D

procedure UPLOAD_NODE_TO_E2E_FOLDER(F, D)
  $\texttt{md} \leftarrow \texttt{server.get\_metadata}(D)$

  ▷ Encrypt file
  $k_F \leftarrow\!\!\$\ \{0,1\}^{128}$
  $n_F \leftarrow\!\!\$\ \{0,1\}^{128}$
  $[F]_{k_F}, \tau_F \leftarrow \texttt{AES-GCM.Enc}(F, k_F, n_F, \emptyset)$

  ▷ If the file name already exists, reuse the ID of that file
  $k_{md}^{(i)} \leftarrow$
  $\texttt{RSA-ECB-OAEPWithSHA-256AndMGF1Padding.Dec}(sk, [k_{md}^{(i)}]_{pk})$
  $\forall (i, \mathcal{K}_i) \in \texttt{md.keys}$ for $[k_{md}^{(i)}]_{pk} \in \mathcal{K}_i$
  $i_{max} \leftarrow |\texttt{md.keys}|$
  $G \leftarrow \{g_m \mid F.name = g_m.name,$
  $\quad g_m \leftarrow \texttt{AES-GCM.Dec}(k_{md}^{(i)}, n_m, \tau_m, [g_m]_{k_{md}^{(i)}}, \emptyset),$
  $\quad \text{and } (id_g, [g_m]_{k_{md}^{(i)}}, n_m, \tau_m, i) \in \texttt{md.files}\}$
  if $|G| = 1$ **then**
    $id_F \leftarrow id_g$
  else
    $id_F \leftarrow\!\!\$\ \mathcal{F}_{\mathcal{ID}}$ ▷ for the set of file identifiers $\mathcal{F}_{\mathcal{ID}}$
  $t \leftarrow \texttt{server.lock}(\texttt{md})$
  $\texttt{server.upload}(t, id_F, [F]_{k_F})$

  ▷ Add file encryption key to E2E folder
  $F_m \leftarrow (k_F, n_F, \tau_F, F.name, F.mimetype, version)$

  $n_m \leftarrow\!\!\$\ \{0,1\}^{128}$
  $[F_m]_{k_{md}^{(i_{max})}}, \tau_m \leftarrow \texttt{AES-GCM.Enc}(k_{md}^{(i_{max})}, F_m, n_m, \emptyset)$
  $\texttt{md.files} \leftarrow \texttt{md.files} \cup \{(id_F, [F_m]_{k_{md}^{(i_{max})}}, n_m, \tau_m, i_{max})\}$
  $\texttt{server.upload\_metadata}(t, \texttt{md})$
  $\texttt{server.unlock}(\texttt{md})$

Algorithm 21 Nextcloud's planned procedure to create an E2EE folder D shared with all recipients in the set $\mathcal{R}$

1: **procedure** CREATE_E2EE_FOLDER(D, $\mathcal{R}$)
2: `server.mark_e2ee`(D)
3: $k_{md}^{(0)} \leftarrow_{\$} \{0,1\}^{128}$
4: $\forall r \in \mathcal{R}.[k_{md}^{(0)}]_{pk_r} \xleftarrow{\$}$ `RSA-ECB-OAEPWithSHA-256AndMGF1Padding.Enc`$(pk_r, k_{md}^{(0)})$
5: `md.keys` $\leftarrow \{(0, \{[k_{md}^{(0)}]_{pk_r} \mid \forall r \in \mathcal{R}\})\}$
6: $\forall r \in \mathcal{R}.[(r, pk_r)]_{k_{md}^{(0)}}, \tau_r \leftarrow$ `AES-GCM.Enc`$(k_{md}^{(0)}, (r, pk_r), n_r, \emptyset)$ ▷ where $n_r \leftarrow_{\$} \{0,1\}^{128}$
7: `md.sharer` $\leftarrow \{([(r, pk_r)]_{k_{md}^{(0)}}, n_r, \tau_r) \mid \forall r \in \mathcal{R}\}$

the public key for s before, it fetches it from the server and stores it in the local keychain. Although the client verifies the server's signature, this only protects against MitM attacks and not malicious providers. After the first exchange, the client uses the locally stored values instead of re-fetching the certificate. Next, it re-encrypts all metadata key versions $k_{md}^{(i)}$ under the public key pk_s of the new recipient s. It is necessary to encrypt all keys since clients re-encrypt file metadata lazily. Therefore, some file keys may still be encrypted with old metadata keys. In the end, the client uploads the updated metadata to the server. Note that the white papers do not mention the locking for this algorithm, and we added it where it would make sense.

Algorithm 23 shows Nextcloud planned – but not yet implemented – design for the reverse process of removing a user s from a shared folder D. In this case, the client r first decrypts all metadata. Next, r generates a new metadata file key $k_{md}^{(i_{max}+1)}$ and re-encrypts it for all former recipients except s. It removes s from the recipient list and encrypts the updated list with $k_{md}^{(i_{max}+1)}$. We again added the metadata file locking ourselves. We note that r does not re-encrypt all file metadata, it only adds a new metadata key and encrypts it for all other sharers and not anymore for the removed user s. Although there is no implementation yet, Nextcloud presumably wants to avoid full re-encryption for performance reasons. The price of this lazy re-encryption is that we need to maintain old metadata key versions.

B.1.5 Account Recovery

Nextcloud offers the option of using a central recovery key. This is an additional asymmetric key, which the user should export and store in a secure location. In addition to the user's public key, the clients transparently encrypt all key material for this recovery key. The white paper mentions that

Algorithm 22 Nextcloud's procedure how a user r shares an E2EE folder D with user s

1: **procedure** SHARE_E2EE_FOLDER(D, s)
2: **if** $\{pk_s, s\}_{sk_{server}} \notin$ `keychain.Get`(s) **then** ▷ TOFU
3: $\{pk_s, s\}_{sk_{server}} \leftarrow$ `server.get_cert`(s)
4: **if not** `X509.Vfy`$(\{pk_s, s\}_{sk_{server}}, pk_{server})$ **then return** $\perp$
5: `keychain.Put`$(\{pk_s, s\}_{sk_{server}})$

▷ Re-encrypt metadata keys for s
6: `md` $\leftarrow$ `server.get_metadata`(D)
7: $t \leftarrow$ `server.lock`$(\texttt{md})$
8: **for all** $(i, \mathcal{K}_i) \in$ `md.keys`, $[k_{md}^{(i)}]_{pk_r} \in \mathcal{K}_i$ **do**
9: $k_{md}^{(i)} \leftarrow$ `RSA-ECB-OAEPWithSHA-256AndMGF1Padding.Dec`$(sk_r, [k_{md}^{(i)}]_{pk_r})$
10: $[k_{md}^{(i)}]_{pk_s} \xleftarrow{\$}$ `RSA-ECB-OAEPWithSHA-256AndMGF1Padding.Enc`$(pk_s, k_{md}^{(i)})$

▷ Update the metadata (keys and sharing)
11: $i_{max} \leftarrow |$`md.keys`$|$
12: `md.keys` $\leftarrow \{(i, \mathcal{K}_i \cup \{[k_{md}^{(i)}]_{pk_s}\}) \mid \forall (i, \mathcal{K}_i) \in$ `md.keys`$\}$
13: $n_s \leftarrow\!\!\$\ \{0,1\}^{128}$
14: $[(s, pk_s)]_{k_{md}^{(i_{max})}}, \tau_s \leftarrow$ `AES-GCM.Enc`$(k_{md}^{(i_{max})}, (s, pk_s), n_s, \emptyset)$
15: `md.sharer` $\leftarrow$ `md.sharer` $\cup \{([(s, pk_s)]_{k_{md}^{(i_{max})}}, n_s, \tau_s)\}$
16: `server.upload_metadata`$(t, \texttt{md})$
17: `server.unlock`$(\texttt{md})$

clients should warn users when such a recovery key exists. However, they do not describe what measures would prevent a malicious cloud provider from removing this notification.

B.2 Observations and Properties

This section briefly discusses our observations on Nextcloud's source code (Appendix B.2.1). Furthermore, we examine the implications of Nextcloud's design for privacy (Appendix B.2.2), confidentiality (Appendix B.2.3), and authentication (Appendix B.2.4).

Algorithm 23 Nextcloud's procedure for user r to remove a user s from a shared E2EE folder D

1: **procedure** UNSHARE_E2EE_FOLDER(D, s)
▷ Decrypt all metadata
2: $\texttt{md} \leftarrow \texttt{server.get\_metadata}(D)$
3: $t \leftarrow \texttt{server.lock}(\texttt{md})$
4: **for all** $(i, \mathcal{K}_i) \in \texttt{md.keys}, [k_{md}^{(i)}]_{pk_r} \in \mathcal{K}_i$ **do**
5: $k_{md}^{(i)} \leftarrow \texttt{RSA-ECB-OAEPWithSHA-256AndMGF1Padding.Dec}(sk_r, [k_{md}^{(i)}]_{pk_r})$
6: $i_{max} \leftarrow |\texttt{md.keys}|$
7: $\mathcal{R} \leftarrow \emptyset$
8: **for all** $([(r, pk_r)]_{k_{md}^{(i)}}, n_r, \tau_r) \in \texttt{md.sharer}$ **do**
9: $(r, pk_r) \leftarrow \texttt{AES-GCM.Dec}(k_{md}^{(i)}, n_r, \tau_r, [(r, pk_r)]_{k_{md}^{(i)}}, \emptyset)$
10: **if** $r \neq s$ **then**
11: $\mathcal{R} \leftarrow \mathcal{R} \cup \{r\}$

▷ Update sharers and encrypt new metadata key for them
12: $k_{md}^{(i_{max}+1)} \leftarrow_{\$} \{0,1\}^{128}$
13: $\forall r \in \mathcal{R}.[k_{md}^{(i_{max}+1)}]_{pk_r} \xleftarrow{\$} \texttt{RSA-ECB-OAEPWithSHA-256AndMGF1Padding.Enc}(pk_r, k_{md}^{(i_{max}+1)})$
14: $\texttt{md.keys} \leftarrow \{(0, \{[k_{md}^{(i_{max}+1)}]_{pk_r} \mid \forall r \in \mathcal{R}\})\}$
15: $\forall r \in \mathcal{R}.[(r, pk_r)]_{k_{md}^{(i_{max}+1)}}, \tau_r \leftarrow \texttt{AES-GCM.Enc}(k_{md}^{(i_{max}+1)}, (r, pk_r), n_r, \emptyset)$ ▷ where $n_r \leftarrow_{\$} \{0,1\}^{128}$
16: $\texttt{md.sharer} \leftarrow \{([(r, pk_r)]_{k_{md}^{(i_{max}+1)}}, n_r, \tau_r) \mid \forall r \in \mathcal{R}\}$
17: $\texttt{server.update\_metadata}(t, \texttt{md})$
18: $\texttt{server.unlock}(\texttt{md})$

B.2.1 Source Code Observations

Nextcloud uses the common OpenSSL library for encryption, which is well known and available on many platforms. Moreover, they use two good encryption schemes: `AES-GCM` and `RSA-ECB-OAEPWithSHA-256AndMGF1Padding`.

They are not very transparent regarding the implemented features. For instance, the website claimed for a while to provide sharing of encrypted folders, which caused some controversy on GitHub [34]. One possible reason for the missing functionality is that the closed source enterprise version of ownCloud had more features than the community edition. Thus, Nextcloud needed to rebuild some functionality after the fork. Another reason is that open-source projects tend to progress slowly because many people are volunteers, and there is a significant communication overhead.

We used the static code analysis tool SonarQube [49] to scan Nextcloud's code for vulnerabilities. For the 391,693 lines of code of the Nextcloud server, the tool reported 453 bugs (1 critical, 319 major, 133 minor), 1 critical vulnerability, and 758 security warnings (56 high, 171 medium, 531 low). The major bugs primarily involve deprecated functions and mistreated return values. The critical vulnerability is a 512-bit `RSA` key, which is only used in a test vector. The high-security warnings mainly concern generous folder permissions, which should be reviewed in more detail. Furthermore, there are many false positives for hardcoded credentials in unit tests. The server-internally used weak hash functions `MD5` and `SHA-1` cause medium warnings. Despite over 10,000 code best practice issues, the code still achieves the best score for maintainability. The client's analysis shows very similar results, as it shares significant parts of its 407,058 lines of code with the server. In conclusion, SonarQube did not discover major vulnerabilities, and Nextcloud's codebase seems to be well-maintained.

B.2.2 Privacy

Nextcloud hides the directory structure, file names, and data in encrypted folders. However, they explicitly accept leaking the number of files in a single folder [146]. Nextcloud does not provide an official GDPR because this product can be self-hosted, and the gathered data, especially usage data, depends on the installation. According to their administration manual [17], they only store session and user cookies, the latter containing user IDs.

As a consequence of the per-folder activation of E2EE, not all content of a user is protected from the service provider. Moreover, the account itself is accessible to Nextcloud as well, since the user sends his password in cleartext for authentication (as discussed in Appendix B.2.4).

B.2.3 Confidentiality

In general, we get the impression that Nextcloud is designed with a self-hosted and therefore trusted service provider in mind. Although they provide E2EE, it is more integrated for additional security in transit or against physical attacks on the server than for protecting against a malicious provider.

A significant threat to confidentiality is that the adversary acts as the root of trust for the certificate signing. Although certificates are stored locally, we still need the TOFU assumption. A malicious provider can always impersonate users on all certificate requests and thus perform a MitM attack against the protocol. As a result, the client unintentionally shares encrypted folders with the adversary instead of another user.

In addition, note that the user removal described in Algorithm 23 only considers lazy re-encryption of the file's metadata keys. Nextcloud does not mention changing the file key when a client updates the file metadata to the new metadata key. Such an implementation has the disadvantage that a user who stored all file keys and later, after being removed from the shared folder, gets access to the ciphertexts can still decrypt the file content. As long as the file key is not changed, removed users can even see newer file versions.

B.2.4 Authentication

Appendix B.1.2 explained that the authentication procedure completely trusts the server as the client sends the user's password in cleartext. However, although any TLS-MitM adversary or malicious cloud can authenticate as the user, they do not receive access to encrypted folders: the secret key for these directories is stored locally in the keychain of the trusted devices. Any authenticated user can request the encrypted secret key but only decrypt it with the randomly generated mnemonic.

As we already discussed in Appendix B.2.3, other users are authenticated by relying on certificates signed by the storage provider. In the case of a malicious provider, this process fails to provide secure authentication. Nextcloud's white paper suggests to use certificate transparency or another logging mechanism in the future [146]. With such an integrity-protected history, clients could detect misbehaving servers.

Furthermore, a TLS-MitM attacker can impersonate the server against a new device. Since the new client does not yet have the server's public key, it requests it. An adversary can insert their public key and consequently generate certificates for other users that the client accepts. This issue is coherent with Nextcloud's reliance on TOFU: the first connection of any device needs to be in a secure environment.

Appendix C

Mega TLS Cipher Suites

We scan the offered TLS cipher suites of three servers of the cloud storage provider Mega with the following nmap command

```
nmap -sV --script ssl-enum-ciphers -p 443 < domain >
```

The servers for the domains g.api.mega.co.nz and mega.nz offer the following TLS cipher suites:

- TLS-1.3:
  - TLS_AKE_WITH_AES_128_GCM_SHA256
  - TLS_AKE_WITH_AES_256_GCM_SHA384
  - TLS_AKE_WITH_CHACHA20_POLY1305_SHA256
- TLS-1.2:
  - TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
  - TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
  - TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
  - TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256
  - TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
  - TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
  - TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
  - TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
  - TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256-draft
  - TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256-draft

We notice that there are cipher suites in TLS-1.2 that use AES-CBC and SHA-256. The former may be vulnerable to the plaintext recovering MitM attacks presented by Paterson et. al. in [76]. The *SHAttered* paper [155] found the first collision on SHA-1 in 2017. This hash function is now considered to be deprecated and should be avoided.

The server for static content at the domain eu.static.mega.co.nz supports older TLS versions and cipher suites:

- TLS-1.3:
  - TLS_AKE_WITH_AES_256_GCM_SHA384
  - TLS_AKE_WITH_CHACHA20_POLY1305_SHA256
  - TLS_AKE_WITH_AES_128_GCM_SHA256
- TLS-1.2:
  - TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
  - TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
  - TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
  - TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
  - TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
  - TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
  - TLS_DHE_RSA_WITH_AES_256_CCM_8
  - TLS_DHE_RSA_WITH_AES_256_CCM
  - TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
  - TLS_DHE_RSA_WITH_AES_256_CBC_SHA
  - TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
  - TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
  - TLS_DHE_RSA_WITH_AES_128_CBC_SHA
  - TLS_RSA_WITH_AES_128_CBC_SHA
  - TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
  - TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256
  - TLS_ECDHE_RSA_WITH_ARIA_256_GCM_SHA384
  - TLS_DHE_RSA_WITH_ARIA_256_GCM_SHA384
  - TLS_DHE_RSA_WITH_AES_128_CCM_8

  - TLS_DHE_RSA_WITH_AES_128_CCM
  - TLS_ECDHE_RSA_WITH_ARIA_128_GCM_SHA256
  - TLS_DHE_RSA_WITH_ARIA_128_GCM_SHA256
  - TLS_ECDHE_RSA_WITH_CAMELLIA_256_CBC_SHA384
  - TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA256
  - TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
  - TLS_ECDHE_RSA_WITH_CAMELLIA_128_CBC_SHA256
  - TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA256
  - TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA
  - TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA
  - TLS_RSA_WITH_AES_256_GCM_SHA384
  - TLS_RSA_WITH_AES_256_CCM_8
  - TLS_RSA_WITH_AES_256_CCM
  - TLS_RSA_WITH_ARIA_256_GCM_SHA384
  - TLS_RSA_WITH_AES_128_GCM_SHA256
  - TLS_RSA_WITH_AES_128_CCM_8
  - TLS_RSA_WITH_AES_128_CCM
  - TLS_RSA_WITH_ARIA_128_GCM_SHA256
  - TLS_RSA_WITH_AES_256_CBC_SHA256
  - TLS_RSA_WITH_CAMELLIA_256_CBC_SHA256
  - TLS_RSA_WITH_AES_128_CBC_SHA256
  - TLS_RSA_WITH_CAMELLIA_128_CBC_SHA256
  - TLS_RSA_WITH_AES_256_CBC_SHA
  - TLS_RSA_WITH_CAMELLIA_256_CBC_SHA
  - TLS_RSA_WITH_CAMELLIA_128_CBC_SHA
- TLS-1.1:
  - TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
  - TLS_DHE_RSA_WITH_AES_256_CBC_SHA
  - TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
  - TLS_DHE_RSA_WITH_AES_128_CBC_SHA

  - TLS_RSA_WITH_AES_128_CBC_SHA
  - TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA
  - TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA
  - TLS_RSA_WITH_AES_256_CBC_SHA
  - TLS_RSA_WITH_CAMELLIA_256_CBC_SHA
  - TLS_RSA_WITH_CAMELLIA_128_CBC_SHA
- TLS-1.0:
  - TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
  - TLS_DHE_RSA_WITH_AES_256_CBC_SHA
  - TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
  - TLS_DHE_RSA_WITH_AES_128_CBC_SHA
  - TLS_RSA_WITH_AES_128_CBC_SHA
  - TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA
  - TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA
  - TLS_RSA_WITH_AES_256_CBC_SHA
  - TLS_RSA_WITH_CAMELLIA_256_CBC_SHA
  - TLS_RSA_WITH_CAMELLIA_128_CBC_SHA

In addition to the concerns discussed above for the more sensitive domains, we note what the static content server includes many cipher suites which do not offer perfect forward secrecy (i.e., they use static RSA keys for key transfer). Moreover, this server still supports the outdated TLS-1.0 protocol.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

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

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

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

Title of work (in block letters):

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

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

With my signature I confirm that

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

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

Place, date	**Signature(s)**

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