



Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

# Breaking Cryptography in the Wild: Nextcloud

Semester Project

Daniele Coppola

January 01, 2023

Advisors: Prof. Dr. K. Paterson, Prof. Dr. M. Albrecht, M. Backendal

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



---

# Contents

---

<b>Contents</b>	<b>i</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>3</b>
2.1 Notation . . . . .	3
2.1.1 Object-oriented Syntax . . . . .	3
2.1.2 Cryptographic Zoo . . . . .	4
2.1.3 Shared objects . . . . .	5
<b>3 Threat Model</b>	<b>7</b>
3.1 Server side encryption . . . . .	7
3.2 End-to-end encryption . . . . .	7
<b>4 Pseudocode</b>	<b>9</b>
4.1 Client authentication . . . . .	9
4.1.1 Token . . . . .	9
4.1.2 Web browser log in . . . . .	11
4.1.3 Desktop client log in . . . . .	11
4.2 End-To-End Encryption (E2EE) . . . . .	13
4.2.1 Key hierarchy . . . . .	14
4.2.2 E2EE initialization . . . . .	15
4.2.3 E2EE client startup . . . . .	15
4.2.4 Folder metadata . . . . .	16
4.2.5 E2EE folder creation . . . . .	17
4.2.6 Folder synchronization . . . . .	17
<b>5 Discovered vulnerabilities</b>	<b>23</b>
5.1 Metadata key insertion . . . . .	23
5.2 Empty metadata keys . . . . .	24
5.3 IV reuse in file update . . . . .	25
<b>6 Mitigation of Attacks on Nextcloud</b>	<b>29</b>
<b>7 Suggestions</b>	<b>31</b>
7.1 Improvements to authentication . . . . .	31
7.2 RSA master key verification . . . . .	32
7.3 Folder sharing . . . . .	33

<b>8</b>	<b>Conclusions</b>	<b>35</b>
<b>A</b>	<b>Appendix</b>	<b>37</b>
A.1	Key recovery attack . . . . .	37
A.2	RSA encryption with multiple primes in the modulus . . . . .	44
A.3	Oracle queries to filter candidate public keys . . . . .	45
	<b>Bibliography</b>	<b>49</b>

## Chapter 1

---

# Introduction

---

Cloud storage is a cloud computing model that enables storing data and files on the internet through a cloud computing provider that one accesses either through the public internet or a dedicated private network connection. Recent studies have shown that 50% of the global data will be stored in the cloud by 2025[16]. Cloud storage solutions concentrate data from multiple users onto a single cloud provider. Consequently, a vulnerability in a major cloud provider will affect millions of users.

Keeping data safe and secure is a central theme for most cloud providers. Some providers only defend against external adversaries trying to gain access to the data. These providers, more or less implicitly, ask their customers to trust them for data confidentiality. Other more progressive cloud providers offer end-to-end encryption solutions that claim to maintain data confidentiality, even in the face of a malicious service provider. They are interesting from a cryptanalysis perspective because the challenging adversarial model presents more opportunity for mistakes to be made in cryptographic design and implementation. In a recent Master's thesis [11], Haller identified a number of such storage services and showed that a prominent one, MEGA, contained significant cryptographic flaws. He also provided an overview of the cryptographic architecture of another popular service, Nextcloud. This project follows Haller's work and provides a detailed analysis of the cryptography used in Nextcloud.

### Nextcloud

Nextcloud is an open source software that allows enterprises and private individuals to host their own cloud providing platform. Estimating the number of users is challenging because they are spread over self-hosted Nextcloud server instances. In 2017, Nextcloud estimated to have “well over 20 million users” [20]. Today, Nextcloud reports more than 400000 deployments [24], therefore we expect the total number of users to be multiple factors higher.

The core focus of Nextcloud is to put the customer in control over their data and ensure that “data meant to stay private will stay private” [22]. Because of the security the service claims to offer, Nextcloud attracts users who wish to protect highly sensitive data. Nextcloud is used at the computer science department of ETH Zurich for hosting sensitive internal documents and by many other organizations such as Amnesty International and the German federal government [23, 18].

Nextcloud is developing an end-to-end encryption (E2EE) module to protect data privacy even in case of a compromised or malicious Nextcloud server. This project will focus on analyzing the current E2EE design and implementation.



## Chapter 2

---

# Background

---

### 2.1 Notation

We define and use the following conventions in this report:

- $x \leftarrow y$ ; sets element  $x$  to the value  $y$ .
- $x \leftarrow \$ X$ ; samples element  $x$  from set  $X$ .
- $r \leftarrow \$ f(x)$ ; indicates a randomized function  $f$ , which in this instance produces  $r$  on input  $x$ .
- $\perp$ ; indicates general errors and algorithm failures. We may use subscripts  $\perp_0, \perp_1 \dots$  to indicate different errors.
- $[m]_k$ ; ciphertext representing the encryption of message  $m$  with key  $k$ .
- $|s|$ ; 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  $b$ .
- $\text{bits}(a, n)$ ; encodes the integer  $a$  in a bit string of length  $n$ .
- $s_1 \parallel s_2$ ; concatenates  $s_1$  with  $s_2$ .
- $s[a : b]$ ; extracts a substring from  $s$  from char/byte  $a$  to char/byte  $b$  (not included).
- $\text{ord}(g, N)$  is the order of an element  $g$  in  $\mathbb{Z}_N^*$ .
- $\text{lcm}_{i=1}^k(x_i)$ ; indicates the least common multiple among values  $x_i$  with  $i$  ranging from 1 to  $k$ .
- $\varepsilon$ ; indicates an empty string.

#### 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(args)`; for the API call of the function `fn` with argument (s) `args` on the server. The returned value has two fields: `code` for the http status code and `content` for the content returned by the server.

- `keychain.Put(id, x)/keychain.Get(id)`; to specify storing/retrieving a value `x` with identifier `id` in/from the on-device keychain.
- `obj.x`; for the variable `x` stored in context of `obj`.

### 2.1.2 Cryptographic Zoo

#### AES

- Cipher Block Chaining (CBC)
  - $c \leftarrow \text{AES-CBC.Enc}(k, m, iv)$ ; encrypts the message `m` using the key `k` and the initialization vector `iv` to obtain the ciphertext `c`.
  - $m \leftarrow \text{AES-CBC.Dec}(k, c, iv)$ ; decrypts the ciphertext `c` using the key `k` and the initialization vector `iv` to obtain the message `m`.
- Galois Counter Mode (GCM)
  - $(c, \tau) \leftarrow \text{AES-GCM.Enc}(k, m, n, ad)$ ; encrypts the message `m` with associated data `ad` using the key `k` and the nonce `n` to obtain the ciphertext `c` and tag  `$\tau$` .
  - $m/\perp \leftarrow \text{AES-GCM.Dec}(k, c, n, ad)$ ; attempts to decrypt the ciphertext `c` with associated data `ad`, nonce `n` and tag  `$\tau$`  using the key `k`. If authentication fails  $\perp$  is returned, the message `m` otherwise.

The tag is obtained evaluating a polynomial in the Galois Field  $\text{GF}(2^{128})$ . The elements of this field are 128 bit strings. The addition is the xor of the operands. The multiplication consists of two phases: (i) carry-less multiplication of the two 128-bit operands, to generate a 256-bit result; (ii) and reduction modulo the irreducible polynomial  $g(x) = x^{128} + x^7 + x^2 + x + 1$ . For those who are more familiar with modular arithmetic, the reduction modulo  $g(x)$  is equivalent to taking the modulus with respect to a prime number.

#### RSA

In the following, we define the abstract notation used in the remaining part of the report.

- $(sk, pk) \leftarrow \$ \text{RSA.Gen}(l)$ ; generates a public key `pk` and the corresponding private key `sk` with an `l`-bit modulus.
- $c \leftarrow \$ \text{RSA.Enc}(pk, m)$ ; 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).
- $m/\perp \leftarrow \text{RSA.Dec}(sk, c)$ ; attempts to decrypt the ciphertext `c` using the secret key `sk`. Returns message `m` on success,  $\perp$  otherwise.

**RSA-OAEP-WITH-SHA-256-AND-MGF1-Padding** 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. OAEP is instantiated with the hash function SHA-256 and the Mask Generation Function MGF1. Unless otherwise specified, `RSA.Enc(pk, m)` and `RSA.Dec(sk, c)` will always use this encoding/decoding.

#### Hash

Hash functions will be called in the following way:



- $\text{digest} \leftarrow \text{HASH}(m)$ ; generates the hash of message  $m$  and stores it in **digest**

## HKDF

The Hashed Message Authentication Code (HMAC)-based key derivation function (HKDF) is defined in the rfc5869[7] and will be called in the following way:

- $k \leftarrow \text{HKDF}(km, s, \text{HASH}, \text{len})$ ; derives key  $k$  of length  $\text{len}$  expressed in bytes, using key material  $km$ , salt  $s$  and the hash function  $\text{HASH}$

### 2.1.3 Shared objects

Two objects are considered to be always available in the pseudocode:

- **db**; this will be used to get/put/remove objects from/in the database. For simplicity, we consider the database a key value store with the following APIs
  - **db.get**( $k, T$ ); returns the value corresponding to the key  $k$  in the table  $T$ . If the key does not exist in the table,  $\perp$  is returned.
  - **db.put**( $k, v, T$ ); inserts the key value pair  $(k, v)$  in table  $T$ .
  - **db.remove**( $k, T$ ); removes the key value pair identified by key  $k$  from table  $T$ .
- **fs**; this object will be used to read and write files in the local file system of the client.



# Threat Model

---

Nextcloud offers two main options for protecting data at rest: server side encryption (SSE) and end-to-end encryption (E2EE). Each option acts under a specific threat model providing different security guarantees.

### 3.1 Server side encryption

Server side encryption provides protection for data on external storage. Files are encrypted on the Nextcloud server before they are sent to the external storage. The keys used for the encryption never leave the Nextcloud server. Each file is encrypted with a unique file key which is encrypted with a server-wide key or a per-user key.

**Threat model.** There are two sub-settings for server-side encryption: server-wide keys and per-user keys. The threat model considered when using a server-wide key consists of an adversary that has full control over the remote storage. The adversary can read, modify and delete the stored files. Nextcloud server and admin are assumed to be trusted.

If per-user keys are used, a stronger adversarial setting is considered. On top having full access to the remote storage, the adversary has access to the data at rest on the Nextcloud server [25]. Every per-user key is encrypted by the user's password and therefore, even if an adversary has access to the data at rest on the Nextcloud server, it cannot decrypt the user's key.

Note that a malicious server administrator could log the users' passwords when they log in and trivially decrypt users' files. As said, SSE does not protect against a fully compromised Nextcloud server.

### 3.2 End-to-end encryption

The Nextcloud end-to-end encryption feature is designed such that the server never has access to unencrypted files or keys, nor does server-provided code ever handle unencrypted data which could provide avenues for compromise. Files are encrypted client-side and then uploaded on the server. The file keys are uploaded encrypted under a secret known to the client only. Moreover, server-signed certificates and a Trust On First Use (TOFU) model protect against attackers trying to impersonate other users.

**Threat model.** According to Nextcloud documentation “*End-to-end Encryption in Nextcloud protects user data against any attack scenario between user devices, even in case of an undetected, long-term security breach or against untrusted server administrators*”. In this threat model the service provider itself is considered potentially adversarial,

and yet the service should remain secure. The adversary has full access to the encrypted data and keys and can also interact with clients via legitimate channels during steps like authentication and file upload.

**Security guarantees.** The following security properties have to be satisfied [19]:

- Access to file and metadata ciphertexts must not leak directory structure nor file names or content.
  - Leaking the number of files in an encrypted folder is an accepted risk.
- Public keys of users must be auditable. This security guarantee is not explained any further in the white paper and hard to interpret. Our best explanation is that users must be able to authenticate other users' public keys.
- Once a user has been removed from an encrypted folder they should have no relevant key material to decrypt files updated or created in the future.
- Encrypted folders must use an encryption scheme fulfilling the following criteria:
  - Confidentiality: no one, except the legitimate recipients, must have access to the encrypted documents.
  - Integrity: Even with write access to the ciphertext one should not be able to tamper with the data.
  - Data origin authentication.

## Chapter 4

---

# Pseudocode

---

### 4.1 Client authentication

One of the advantages of cloud storage is that data stored on the cloud can be accessed from any device as long as a connection to the cloud is available. In order to log in to their account and access their stored files, a user must authenticate the client which they use to connect to the server.

In the default version of Nextcloud server, accounts are registered by the server administrator. Nextcloud also offers the option to enable users to register a new account themselves. Every user account consists of a unique user identifier `uid` in the form of a username and a password. The server stores the username of each user and the hashed password in its database. A user can use multiple clients to access the remote storage. Each client is identified by a client identifier `cid` which consists of the user identifier `uid` and a client name picked by the client based on the host machine it is running on.

There are two default login routines, one for the web browser client and one used by the desktop client. Nextcloud also supports two-factor authentication and OAuth [8]. This section focuses on the default log in routines.

The goal of a login procedure is to authenticate the client. Once the client is authenticated the server returns the client a token. Tokens are used by the server to authenticate future requests of the client.

We start by the describing in more details what tokens are, how they are created and used. We then present the login routine used to authenticate from a web browser. Finally, we describe the login routine used by a desktop client. In the pseudocode that follows, the server database is accessed through high-level calls to the object `db`. The syntax for this object was defined in the background section 2.1.3.

#### 4.1.1 Token

Tokens are byte strings sampled randomly by the server for a client. The server uses tokens to encrypt values such as users' passwords or other tokens. The resulting ciphertext is stored in the server database, and the token is sent to the client. Since the token is not stored on the server, an adversary who gets access to the database would not be able to retrieve the values encrypted with the tokens.

This section describes how tokens are used to encrypt and decrypt a value that we will generally call `ptxt`. In practice, `ptxt` corresponds to either the user's password or to another token.

**Figure 4.1:** Encryption and decryption procedures used with tokens.

```

1: procedure ENCRYPT(token, ptxt)
2:   sk, pk  $\leftarrow$  $RSA.Gen(2048)
3:                                      $\triangleright$  Encrypt the private key
4:   keyMaterial  $\leftarrow$  HKDF(token, salt= $\varepsilon$ , SHA512, len=64)
5:   encKey  $\leftarrow$  PBKDF2-SHA1(keyMaterial[: 32], salt="salt", iter=1000, len=16)
6:   tagKey  $\leftarrow$  SHA512(keyMaterial[32 :] || "a")
7:   iv  $\leftarrow$  $ {0, 1}128
8:   ctxtSk  $\leftarrow$  AES-CBC.Enc(encKey, PKCS7.pad(sk), iv)
9:   tagSk  $\leftarrow$  HMAC(tagKey, ctxt || iv)
10:  encSk  $\leftarrow$  ctxtSk || "|" || iv || "|" || tagSk
11:                                      $\triangleright$  Encrypt ptxt
12:  ctxt  $\leftarrow$  $RSA.Enc(pk, ptxt)
13:  return (encSk, ctxt)

1: procedure DECRYPT(token, encSk, ctxt)
2:                                      $\triangleright$  Decrypt the private key
3:  ctxtSk, iv, tagSk  $\leftarrow$  split(encSk, "|")
4:  keyMaterial  $\leftarrow$  HKDF(token, salt= $\varepsilon$ , SHA512, len=64)
5:  encKey  $\leftarrow$  PBKDF2-SHA1(keyMaterial[: 32], salt="salt", iter=1000, len=16)
6:  tagKey  $\leftarrow$  SHA512(keyMaterial[32 :] || "a")
7:  tag  $\leftarrow$  HMAC(tagKey, ctxtSk || iv)
8:  if tag  $\neq$  tagSk then
9:    return  $\perp$ 
10:  sk  $\leftarrow$  PKCS7.unpad(AES-CBC.Dec(encKey, ctxtSk, iv))
11:  return RSA.Dec(sk, ctxt)

```

The ENCRYPT function receives as input a token and a plaintext value `ptxt`. The procedure starts by generating an RSA key pair, (`sk`, `pk`). From the token, an encryption key, `encKey` and a tagging key, `tagKey` are derived and used to encrypt and tag the secret key. The plaintext is encrypted with the RSA key using RSA-OAEP. The returned value consists of the encrypted secret key and the encryption of the plaintext. The DECRYPT function is the inverse of the ENCRYPT function. It receives as inputs a token, the encryption of a secret key, `encSk`, and a ciphertext, `ctxt`. The encryption key and tagging key are derived from the token and used to decrypt the secret key. The secret key is then used to decrypt the ciphertext. If decryption succeeds, the plaintext is returned. Figure 4.1 shows the pseudocode for the ENCRYPT and DECRYPT functions.

We define two additional functions ENCRYPTANDPUT and GETANDDECRYPT to simplify the login diagrams. These are wrappers for the functions ENCRYPT and DECRYPT and handle the storage of the encrypted values in the database.

The function ENCRYPTANDPUT receives as input a token `token`, a plaintext `ptxt` and a client identifier `cid`. The input token can be empty, and, in this case, a new token is sampled inside the routine. The token is used to encrypt the plaintext with the function ENCRYPT. The resulting ciphertext is saved in the server database with the client identifier and a value obtained hashing the token with a salt. The function terminates returning the token.

The function GETANDDECRYPT receives as input a client identifier `cid` and a token `token`. From the database, the function fetches the ciphertext identified by the client identifier and the hash of the token. The ciphertext is decrypted with the function DECRYPT

**Figure 4.2:** Helper functions `ENCRYPTANDPUT` and `GETANDDECRYPT`. They handle the insertion and retrieval of ciphertexts from the database. The value `salt` is a value sampled once when the server is set up, stored in a configuration file and used as a salt when hashing tokens.

```

1: procedure ENCRYPTANDPUT(token, ptxt, cid)
2:   if token =  $\varepsilon$  then
3:     token  $\leftarrow$  $ [a-zA-Z0-9]128
4:   tokenHash  $\leftarrow$  SHA512(token || salt)
5:   (encSk, ctxt)  $\leftarrow$  ENCRYPT(token, ptxt)
6:   db.put((tokenHash, cid), (encSk, ctxt), oc_authtoken)
7:   return token

1: procedure GETANDDECRYPT(token, cid)
2:   tokenHash  $\leftarrow$  SHA512(token || salt)
3:   (encSk, ctxt)  $\leftarrow$  db.get((tokenHash, cid), oc_authtoken)
4:   ptxt  $\leftarrow$  DECRYPT(token, encSk, ctxt)
5:   return ptxt

```

and returned. Figure 4.2 shows the pseudocode of functions `ENCRYPTANDPUT` and `GETANDDECRYPT`.

#### 4.1.2 Web browser log in

In order to log in, the web browser client sends a POST request at the `/login` endpoint containing the username and password in clear text. The server compares the hash of the received password with the one stored in the database using function `CHECKPASSWORD` showed in figure 0. If the check is successful, the server responds with a token that the client will include in its subsequent requests. The following requests are authenticated by the server verifying the attached token. The token and the client identifier are passed as input to the `GETANDDECRYPT` function which returns the user's password. Given the user's password, the function `CHECKPASSWORD` can be used again to authenticate the request.

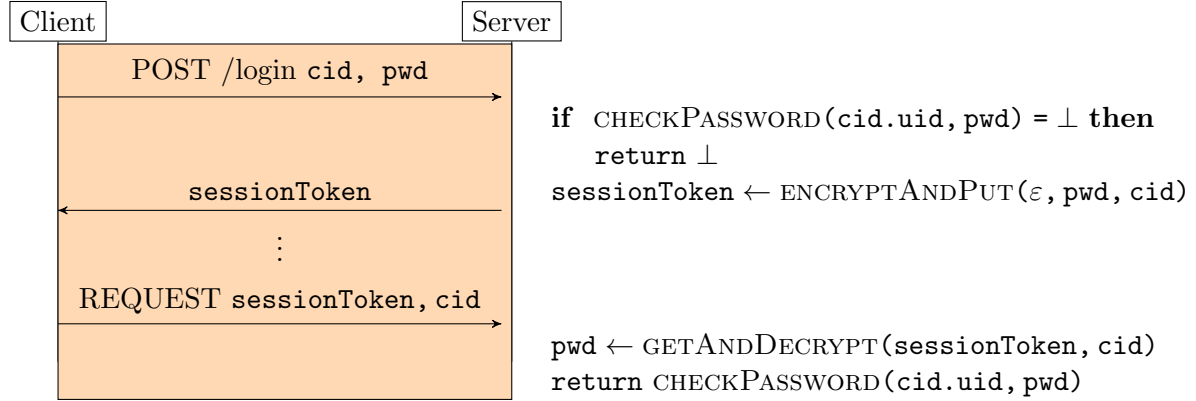
Passwords are hashed using the PHP function call `password.hash` which supports three algorithms: `bcrypt`[27], `argon2i` and `argon2id` (default)[2]. All these algorithms are specifically designed for password hashing and make use of a salt. The php function `password.verify` is used to verify the received password. Based on the hash that the server has stored in the database, the function picks the correct hashing algorithm and compares the password hash with the stored one.

#### 4.1.3 Desktop client log in

In order to authenticate a new desktop client, a user must grant access to the client from an already authenticated web browser session. The desktop client redirects the user to the web browser client from which the user grants access to the desktop client. The desktop client periodically polls the server to checks if it was granted access. Once access is granted, the server sends a token to the desktop client. This token encrypts the user's password and can be used to authenticate the client's requests.

The procedure used by the desktop client starts with an empty POST request to which the server answers with two tokens: the `pollToken` and the `loginToken`. The first is used by the desktop client to poll the server to check if it was granted access. The latter is used

**Figure 4.3:** Messages exchanged between client and server during login. The client sends the client identifier `cid` and the user's password `pwd`. If the login succeeds, the server responds with a token named `sessionToken`. The following requests are authenticated retrieving the user's password with the `sessionToken` and checking the password again.



**Figure 4.4:** Credentials verification

**Input:** user identifier, `uid`; user's password, `pwd`.

**Output:** boolean indicating if password verification succeeded.

- 1: **procedure** CHECKPASSWORD(`uid`, `pwd`)
- 2:   `hash` ← `db.get(uid, oc.users)`
- 3:   **if** !`hash` **then**
- 4:     **return** ⊥
- 5:   **return** `password_verify(pwd, hash)`

in the web browser to access the page `/login/v2/flow/loginToken`. The purpose of the `loginToken` is to tie the login of the desktop client to the log in of the web browser. These tokens are stored together in the server database in the table `loginFlow`. From the web browser, the user logs in as described above and grants access to the device. Once access is granted, the server executes the function `ENCRYPTANDPUT` which creates a token called `appPwd` and uses it to encrypt the user's password. The token `pollToken` is retrieved from the database and used to encrypt the `appPwd`. At this point, the server deletes the entry (`loginToken`, `pollToken`) from the database. The idea is that once a token is used to encrypt a value, the token itself should not be stored on the server. When the desktop client polls the `/poll` end point, the server can use the attached `pollToken` to retrieve the `appPwd` which is returned to the client. This concludes the desktop client log in process. At the end of this process:

- the server stores in its database the user's password encrypted under the `appPwd` token.
- The client stores the token `appPwd`.

Every request from the desktop client will include the token `appPwd`. The server can use the `appPwd` and the client identifier as input to the function `GETANDDECRYPT` to retrieve the user's password. The server can then authenticate the request by checking the user's password with the function `CHECKPASSWORD`.

Both the `sessionToken` used by the web browser and the `appPwd` token are byte strings generated by the server and used to encrypt the user's password. The main difference is their time to live (TTL). The `sessionToken` as the name suggests have a shorter TTL and is refreshed every session, whereas the `appPwd` is a long term token that stays valid



forever unless the user decides to revoke it.

Token revocation is a feature that allows users to revoke access to a specific client. A user can request the server to revoke access to one of its clients specifying the client identifier. The server will remove all the entries in the table `oc_authtoken` corresponding to that particular client identifier.

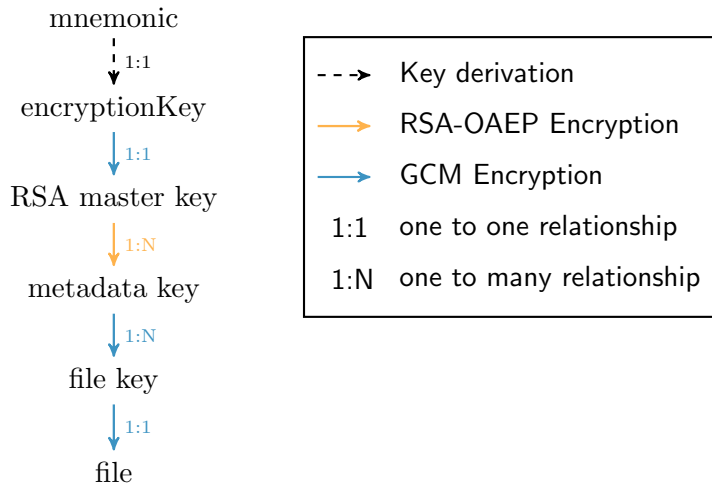
Tokens are stored on the clients using Qt keychain API [1] in the keychain of the device. On Mac OS X passwords are stored in the OS X Keychain. On Linux/Unix if running, GNOME Keyring is used. Since Windows does not provide a service for secure storage QtKeychain uses the Windows API function `CryptProtectData` to encrypt the password with the user's login credentials.

**Figure 4.5:** Desktop client login. The blue sections show the messages exchanged between the desktop client and the server. The orange section shows the messages exchanged between the browser and the server. The web client is identified by the identifier `cid`, and the desktop client is identified by the identifier `cidD`



## 4.2 End-To-End Encryption (E2EE)

Nextcloud provides E2EE to protect users' data even against malicious server administrators or compromised providers. This feature is not available by default in Nextcloud and has to be enabled by the server administrator. Once the feature is enabled, users can choose which folders to protect with E2EE. Folders that are end-to-end encrypted are only accessible through desktop clients. This is a reasonable decision considering the E2EE threat model. If E2EE data was accessible through web browser clients, the code handling such data would be provided by the server which, in the considered setting, can be adversarial and could trivially gain access to the data. The Nextcloud desktop clients

**Figure 4.6:** Nextcloud's key hierarchy

are clients that synchronize the contents of local directories from computers, tablets, and handheld devices to the Nextcloud server. The E2EE module is essentially a layer on top of the synchronization layer that encrypts the files before they are uploaded to the server.

We start by presenting the key hierarchy used by Nextcloud and continue explaining how the previously presented keys are generated and stored. Finally, we illustrate the synchronization process and provide the pseudocode for file upload and file download.

#### 4.2.1 Key hierarchy

This section provides a bottom-up description of the key hierarchy. Figure 4.6 shows a visual representation of it.

At the bottom of the hierarchy are file keys. File keys are used to encrypt files and are freshly generated by the client for every new file. Every folder is associated with a metadata key that the client generates when a folder is marked as E2EE. The metadata key of a folder is used to encrypt the file keys for the files contained in that folder. Every user has an RSA master key used to encrypt metadata keys. The private part of the master key is encrypted with an encryption key derived from a 12 words mnemonic which stands at the root of the key hierarchy. The 12 words mnemonic is unique per user, and it is generated by the client sampling randomly from a set of 2048 words. The user is strongly recommended to record these somewhere secure as the complete loss of this private key means there is no way to access their data anymore.

All the presented keys, except for the mnemonic and the public part of the RSA master key, are stored encrypted on the server to support access from multiple devices. A user on a new device, after authenticating to the server, can enter the mnemonic, fetch the encrypted key material from the server and decrypt it. Once file keys are recovered, the encrypted files can be fetched from the server and decrypted. The public part of the RSA master key is stored in plaintext on the server.

Note that the 12 words mnemonic is independent of the user password. As a consequence, client authentication and access to the E2EE folders are kept separated. This is a praiseworthy design choice because the recovery of users' password will not hinder the security of the E2EE files.

### 4.2.2 E2EE initialization

In the previous section we explained how desktop login works. Here we assume that the client has already logged in with user identifier `uid`. The initialization procedure is executed once per user when E2EE is enabled.

When the user enables E2EE the client generates a 12 word mnemonic and the RSA master key as shown in 1. The server provides the client with a certificate that binds the client's public key to its user identifier. Certificates can be used by other users to authenticate each other. However, since sharing is not implemented, the relevance of this certificate is marginal at the time of our analysis. The mnemonic and the master key are stored in the keychain of the device. After encrypting the user's private key with an encryption key derived from the mnemonic, the master key is uploaded to the server.

To briefly summarize the state at the end of this procedure:

- the client's keychain stores the mnemonic and the RSA master key.
- The server stores the users's certificate and the encrypted private key in a specific folder that collects certificates and keys of all the users.

---

**Algorithm 1** E2EE initialization

---

**Input:** User identifier, `uid`; object to make requests to the server, `server`.

**Output:** RSA master key, `sk`, `pk`; mnemonic of the user, `mnemonic`.

---

```

1: sk, pk  $\leftarrow$  RSA.Gen(2048) ▷ Generate Keys
2: csr  $\leftarrow$  x509.csr(pk, uid)
3: cert  $\leftarrow$  server.signCsr(csr) ▷ Store keys
4: m  $\leftarrow$  $W12 //  $|W| = 2048$ 
5: s  $\leftarrow$  {0, 1}320 // salt for key derivation
6: k  $\leftarrow$  PBKDF2-SHA1(m, s, iter = 1024, len = 256)
7: n  $\leftarrow$  {0, 1}128 // NONCE for the encryption of sk
8: [sk]k,  $\tau$   $\leftarrow$  AES-GCM.Enc(k, sk, n, ad =  $\varepsilon$ )
9: keychain.Put("certificate", cert)
10: keychain.Put("sk", sk)
11: keychain.Put("mnemonic", m)
12: server.putKey(([sk]k, n,  $\tau$ , s))

```

---

### 4.2.3 E2EE client startup

During client startup, the RSA master key is loaded either from the keychain, if present, or it is fetched from the server. The client master key can then be used to decrypt metadata keys, file keys and consequently files.

As previously mentioned, the public key is stored in plaintext on the server and is not integrity-protected. The client verifies the public key by checking that RSA correctness holds for a randomly sampled message as shown in lines 15-18 of algorithm 2. This verification procedure relies on non-standard standard properties of the RSA-OAEP and is better analyzed in section 7.2.

If the verification succeeds, at the end of the setup algorithm, the client knows the RSA master key of the user and can access the E2EE data.

**Algorithm 2** Setup E2EE

---

**Input:** object used to make requests to the server, **server**; user identifier, **uid**.  
**Output:** RSA master key, **pk**, **sk**; user's mnemonic, **mnemonic**.

If the check on **pk**, **sk** is successful, the output is saved in the keychain.

```

1: cert ← keychain.Get("certificate")
2: sk ← keychain.Get("sk")
3: m ← keychain.Get("mnemonic")
4: if cert ≠ ⊥ AND sk ≠ ⊥ AND m ≠ ⊥ then
5:   return cert.pk, sk, m // master key and mnemonic are available in the keychain
6: cert ← server.getCertificate(uid)
7: pk ← cert.pk
8: [sk]k, n, τ, s ← server.getPrivateKey() // encrypted sk, nonce, tag, salt
9: action ← "check"
10: while action = "check" do
11:   m ← user_input("Enter mnemonic : ")
12:   k ← PBKDF2-SHA1(m, s, iter=1024, len=256)
13:   sk ← AES-GCM.Dec(k, [sk]k, n, ε)
14:                                     ▷ Verify secret key
15:   r ←$ {0, 1}512
16:   [r]pk ← RSA.Enc(pk, r)
17:   r' ← RSA.Dec(sk, [r]pk)
18:   if r = r' then // save the verified master key and mnemonic in the keychain
19:     keychain.Put("certificate", cert)
20:     keychain.Put("sk", sk)
21:     keychain.Put("mnemonic", m)
22:     break
23:   else
24:     sk ← ⊥
25:     m ← ⊥
26:   action ← user_input("Action : [check|cancel]")
27: return pk, sk, m

```

---

**4.2.4 Folder metadata**

For each E2EE folder the client stores on the server the encrypted content of the files and an additional file containing the folder metadata. The folder metadata contains the key material necessary to access files in the folder and is stored encrypted on the server.

At a high level, when a client wants to access the content in an E2EE folder, it first fetches the file containing the encrypted folder metadata, decrypts it and retrieves the key material necessary to access the files. The client synchronizes the content of its local folder with the remote one and finally uploads the folder metadata back to the server.

We start by describing in more detail the folder metadata. We then present the pseudocode used to encrypt and the decrypt the folder metadata.

The metadata of a folder consists of two sets. A set  $\mathcal{K}$  of metadata keys and a set  $\mathcal{F}$  of files' metadata. The set  $\mathcal{K}$  contains tuples of the form  $(i, k_{md}^{(i)})$  where  $k_{md}^{(i)}$  is a metadata key and  $i$  is an integer used to identify the key in the set. Metadata keys encrypt sensitive information of files' metadata, namely the filename and the file key used to encrypt the file.

For each file in the folder, the set  $\mathcal{F}$  contains a tuple of the form  $(obfName, filename, k_f, iv_f, \tau_f)$ .

**Figure 4.7:** Example of an encrypted folder metadata. Metadata keys are encrypted the user's public key  $pk$ . Files' metadata are encrypted with the latest metadata key. In the example, there are two metadata keys and multiple files' metadata, although only one is shown in full.

Encrypted folder metadata	
$\mathcal{K}_e$ :	$(1, [k_{md}^{(1)}]_{pk})$ $(2, [k_{md}^{(2)}]_{pk})$
$\mathcal{F}_e$ :	$(obfName, [filename, k_f]_{k_{md}^{(2)}}, 2, iv_f, \tau_f)$ $\vdots$

We now explain the meaning of each field. Encrypted files are stored on the server under an obfuscated name generated randomly by the client independently of the original filename, **filename**. The values **obfName** and **filename** are used to map obfuscated names with original file names. Files are encrypted using AES-GCM, the values  $k_f$ ,  $iv_f$  and  $\tau_f$  are the file key and the initialization vector used to encrypt the file and the tag returned by the encryption. A tuple stored in  $\mathcal{F}$  contains all the necessary information to access an encrypted file. The mapping between filename and obfuscated name is used by the client to know which encrypted file to fetch from the server. The file key, the initialization vector and the tag are then used to decrypt the file.

Clients store folders metadata encrypted on the server. Metadata keys are encrypted with RSA-OAEP under the user's master key. The metadata key with the highest index is then used to encrypt the sensitive section of each file metadata: the filename and the file key. The index corresponding to the metadata key used for the encryption is saved with the encrypted file metadata. The encrypted metadata keys and files metadata are collected in the sets  $\mathcal{K}_e$  and  $\mathcal{F}_e$ . The full metadata encryption procedure is shown in algorithm 3. Algorithm 4 describes the inverse procedure, used to decrypt the encrypted folder metadata fetched from the server.

Nextcloud's design of the E2EE module allows folder sharing. The algorithms support multiple metadata keys to allow a user to remove another user from a shared folder. According to Nextcloud's whitepaper, all the users with access to a shared folder know the latest metadata key. If a user wants to remove another user from the share, a new metadata key is generated and shared with all the other users except the one removed. More information on the sharing feature can be found in section 7.3. The sharing feature has not yet been implemented. In practice, the set of metadata keys only ever contains one key.

#### 4.2.5 E2EE folder creation

Once E2EE is enabled, empty folders can be marked as E2EE. Algorithm 5 initialize the folder metadata. After initialization, the folder metadata consists of a single metadata key. The set of files' metadata is empty.

#### 4.2.6 Folder synchronization

Nextcloud desktop clients are synchronization clients that keep a local folder synchronized with the remote one on the server. This section starts by describing how file versioning works, namely how the versions of local files are compared with the remote ones to de-

**Algorithm 3** Encryption of folder metadata**Input:** user's public key,  $\text{pk}$ ; set of metadata keys,  $\mathcal{K}$ ; set of files' metadata,  $\mathcal{F}$ .**Output:** set of encrypted metadata keys,  $\mathcal{K}_e$ ; set of encrypted files' metadata,  $\mathcal{F}_e$ .

```

1: procedure ENCRYPTMETADATA( $\text{pk}, \mathcal{K}, \mathcal{F}$ )
2:                                      $\triangleright$  Metadata keys encryption
3:    $\mathcal{K}_e \leftarrow \{\}$ 
4:   for  $(i, k_{\text{md}}^{(i)})$  in  $\mathcal{K}$  do
5:      $[k_{\text{md}}^{(i)}]_{\text{pk}} \leftarrow \text{RSA.Enc}(\text{pk}, k_{\text{md}}^{(i)})$ 
6:      $\mathcal{K} \leftarrow \mathcal{K} \cup (i, [k_{\text{md}}^{(i)}]_{\text{pk}})$ 
7:                                      $\triangleright$  Pick the most recent metadata key
8:    $i^{(\text{max})} \leftarrow |\mathcal{K}|$ 
9:    $k_{\text{md}} \leftarrow k \mid (i^{(\text{max})}, k) \in \mathcal{K}$ 
10:                                      $\triangleright$  Files' metadata encryption
11:    $\mathcal{F}_e \leftarrow \{\}$ 
12:   for  $(\text{obfName}, \text{filename}, k_f, \text{iv}_f, \tau_f)$  in  $\mathcal{F}$  do
13:      $\text{iv}_{\text{md}} \leftarrow \$\{0, 1\}^{128}$  // iv for file metadata encryption
14:      $[(\text{filename}, k_f)]_{k_{\text{md}}}, \tau_{\text{md}} \leftarrow \text{AES-GCM.Enc}(k_{\text{md}}, (\text{filename}, k_f), \text{iv}_{\text{md}}, \varepsilon)$ 
15:      $\text{encrypted} \leftarrow [(\text{filename}, k_f)]_{k_{\text{md}}} \parallel \tau_{\text{md}} \parallel " \parallel \text{iv}_{\text{md}}$ 
16:      $\mathcal{F}_e \leftarrow \mathcal{F}_e \cup (\text{obfName}, \text{encrypted}, i^{(\text{max})}, \tau_f, \text{iv}_f)$ 
17:   return  $\mathcal{K}_e, \mathcal{F}_e$ 

```

**Algorithm 4** Decryption of folder metadata**Input:** user's secret key,  $\text{sk}$ ; set of encrypted metadata keys  $\mathcal{K}_e$ , set of encrypted files' metadata  $\mathcal{F}_e$ .**Output:** set of metadata keys,  $\mathcal{K}$ ; set of files' metadata,  $\mathcal{F}$ 

```

1: procedure DECRYPTMETADATA( $\text{sk}, \mathcal{K}_e, \mathcal{F}_e$ )
2:                                      $\triangleright$  Metadata keys decryption
3:    $\mathcal{K} \leftarrow \{\}$ 
4:   for  $(i, [k_{\text{md}}^{(i)}]_{\text{pk}})$  in  $\mathcal{K}_e$  do
5:      $k_{\text{md}}^{(i)} \leftarrow \text{RSA.Dec}(\text{sk}, [k_{\text{md}}^{(i)}]_{\text{pk}})$ 
6:      $\mathcal{K} \leftarrow \mathcal{K} \cup (i, k_{\text{md}}^{(i)})$ 
7:                                      $\triangleright$  Files metadata decryption
8:    $\mathcal{F} \leftarrow \{\}$ 
9:   for  $(\text{obfName}, \text{encrypted}, i, \tau_f, \text{iv}_f)$  in  $\mathcal{F}_e$  do
10:     $[(\text{filename}, k_f)]_{k_{\text{md}}} \parallel \tau_{\text{md}} \parallel " \parallel \text{iv}_{\text{md}} \leftarrow \text{encrypted}$ 
11:     $k_{\text{md}} \leftarrow k_{\text{md}} \mid (i, k_{\text{md}}) \in \mathcal{K}$ 
12:     $(\text{filename}, k_f) \leftarrow \text{AES-GCM.Dec}(k_{\text{md}}, [(\text{filename}, k_f)]_{k_{\text{md}}}, \text{iv}, \varepsilon)$ 
13:     $\mathcal{F} \leftarrow \mathcal{F} \cup (\text{obfName}, \text{filename}, k_f, \text{iv}_f, \tau_f)$ 
14:   return  $\mathcal{K}, \mathcal{F}$ 

```

**Algorithm 5** E2EE folder setup**Input:** public key of the user,  $\text{pk}$ ; name of the folder,  $\text{foldername}$ .

```

1: procedure CREATEE2EEFOLDER( $\text{pk}, \text{foldername}$ )
2:    $k_{\text{md}} \leftarrow \$\{0, 1\}^{128}$ 
3:    $\mathcal{K} \leftarrow \{(1, k_{\text{md}})\}$ 
4:    $\mathcal{F} \leftarrow \{\}$ 
5:    $\mathcal{K}_e, \mathcal{F}_e \leftarrow \text{ENCRYPTMETADATA}(\text{pk}, \mathcal{K}, \mathcal{F})$ 
6:   server.putMetadata( $\text{foldername}, (\mathcal{K}_e, \mathcal{F}_e)$ )

```

**Figure 4.8:** This table sums up the most relevant combinations of etag's and modifications time with the corresponding operation executed by the client for the specific file. The superscripts  $(\cdot)^l$ ,  $(\cdot)^{fs}$ ,  $(\cdot)^r$  indicate that the information comes from the local database, the file system or the remote folder. The values  $m_0$  and  $m_1$  are two different modification times, and similarly  $e_0$  and  $e_1$  are two different etag's. Finally,  $\varepsilon$  indicates that the specific value is not available. For example, if a file was removed on the server, the server will not return any information about that file and the remote etag is set to  $\varepsilon$ .

$m^l$	$m^{fs}$	$e^l$	$e^r$	operation
$m_0$	$m_0$	$e_0$	$e_0$	none
$m_0$	$m_1$	$e_0$	$e_0$	upload
$m_0$	$m_0$	$e_0$	$e_1$	download
$m_0$	$m_1$	$e_0$	$e_1$	conflict
$m_0$	$m_1$	$e_0$	$\varepsilon$	upload
$m_0$	$m_0$	$e_0$	$\varepsilon$	delete local
$m_0$	$\varepsilon$	$e_0$	$e_0$	delete remote

termine which version is the most recent. Depending on whether the local or the remote copy of a file is more recent, the client downloads or uploads the file. The download and upload procedure are described at the end of the section.

For each file in the folder, clients save in their database the modification time **mtime** and a randomly generated value called **etag** that is updated every time a file changes. The former is used to determine if a file was modified locally, while the latter is used to compare local and remote versions.

During a sync run the client must first detect if one of the two folders have changed files. On the local folder, the client traverses the file tree and compares the modification time of each file with an expected value stored in its database. If the value is not the same, the client determines that the file has been modified in the local repository. The server provides for each file in the remote folder the corresponding **etag**'s. The client compares the **etag** of each file with its expected value. Again, the expected **etag** value is queried from the client database. If the **etag** is the same, the file has not changed and no synchronization occurs. In the event that a value has changed both on the local and on the remote serve, a conflict case is created, both versions are saved locally, and the user is responsible to resolve the conflict. Table 4.8 sums up the operations that the client performs for each file depending on the **etag** and on the modification time.

Folder synchronization are frequently performed by a client and are also triggered if a local file is modified. In the first part of a sync run the client collects the versioning attributes for all files. Based on the recovered attributes, an operation is selected and executed. For example, if a file was modified remotely (different etags), but was not modified locally (same modified time), the remote version is downloaded.

The function **SYNCHRONIZE** describes in more details how a sync run works. It receives as inputs the user's master key  $(sk, pk)$ , the local versioning information **local**, the remote versioning information **remote** and the encrypted folder metadata  $(K_e, \mathcal{F}_e)$ . The object **local** is a set of tuples of the form  $(filename, e^l, m^l, m^{fs})$ . These fields are respectively the name of the file, the etag and the modification time stored in the local database and the modification time retrieved from the file system. The object **remote** is slightly more complicated. Files on the server are stored under an obfuscated name **obfName** and the server does not know the original filename. For this reason, the object **remote**, which the client receives from the server, is a set of tuples of the form  $(obfName, e^r)$  which maps obfuscated names to the etag values saved in the remote database. After decrypting the folder metadata, the client learns the mapping between obfuscated names and filenames.

This mapping is used to associate the remote tuples with the corresponding filename. The extended tuples are saved in a set referred as **remote'**. The two sets **remote'** and **local** are joined with the function **outerJoin**. This function performs an outer join of the two sets using the filename as the common field. If a specific filename **filename** is present in both sets, the two corresponding tuples are merged into a single one of the form  $(\text{filename}, \text{obfName}, m^l, m^{fs}, e^l, e^r)$ . If a **filename** is present in only one of the two sets, then the attributes coming from the other sets are set to  $\varepsilon$ . For each file an operation is selected and executed. In the pseudocode, the function **getOperation** receives as input the etags and modification times of a file and returns the function executing the operation selected according to table 4.8. The pseudocode does not show the updates to the remote and local database. If a file is uploaded to the server, the remote database stores a new etag for that file. If a file was locally modified or downloaded, the time of the operation is registered in the local database in the **mtime** field. At the end of the sync run, the folder metadata is re-encrypted and returned. The encrypted folder metadata is uploaded to the server. Algorithm 6 contains the pseudocode for the sync run. In the remaining part of this section we describe the download and upload operations. Local and remote delete simply delete the files locally or remotely. In addition, in case of a remote delete, the client removes the file metadata corresponding to the deleted file from the folder metadata.

---

**Algorithm 6** Folder synchronization

---

```

procedure SYNCHRONIZE(sk, pk, local, remote,  $\mathcal{K}_e$ ,  $\mathcal{F}_e$ )
   $(\mathcal{K}, \mathcal{F}) \leftarrow \text{DECRYPTMETADATA}(\text{sk}, \mathcal{K}_e, \mathcal{F}_e)$ 
   $\triangleright$  Map obfuscated names to original filenames

  remote'  $\leftarrow \{\}$ 
  for (obfName,  $e^r$ ) in remote do
    filename  $\leftarrow \text{filename} \mid (\text{obfName}, \text{filename}, \_, \_, \_) \in \mathcal{F}$ 
    remote'  $\leftarrow \text{remote}' \cup \{(\text{filename}, \text{obfName}, e^r)\}$ 
  joined  $\leftarrow \text{outerJoin}(\text{local}, \text{remote}')$ 
   $\triangleright$  Select and execute an operation for each file
  for (filename, obfName,  $m^l$ ,  $m^{fs}$ ,  $e^l$ ,  $e^r$ ) in joined do
    operation  $\leftarrow \text{getOperation}(m^l, m^{fs}, e^l, e^r)$ 
     $\mathcal{F} \leftarrow \text{operation}(\mathcal{F}, \text{filename}, \text{obfName})$ 
  return  $\text{ENCRYPTMETADATA}(\text{pk}, \mathcal{K}, \mathcal{F})$ 

```

---

**File upload.** If the client determines that its version of a file is more recent than the one on the server, the local version is uploaded to the server executing algorithm 7. The function **UPLOAD** receives as input the set of files' metadata  $\mathcal{F}$ , the filename **filename** and the obfuscated name **obfName**. The last input may be set to  $\varepsilon$  if the file is new and therefore the server does not have a copy of it yet. In case of a file update, obfuscated name, file key and initialization vector are retrieved from the file metadata corresponding to the updated file. If the file is new, the client generates a new file key, initialization vector and obfuscated name. In both cases, the file content is read from the file system with the function **fs.read** and is then encrypted using AES-GCM. The encrypted file is uploaded to the server under the name **obfName**. Finally, the files' metadata is updated and returned.

From the server perspective, the encrypted file is a regular file with name equal to the obfuscated name. The mapping between obfuscated names and original names is encrypted in the folder metadata. The server cannot retrieve the original file name, nor its content.



**Algorithm 7** File upload to E2EE folder

**Input:** set of files' metadata,  $\mathcal{F}$ ; the filename, `filename`; the obfuscated name `obfName`.

**Output:** updated set of files' metadata  $\mathcal{F}$

```

1: procedure UPLOAD( $\mathcal{F}$ , filename, obfName)
2:                                      $\triangleright$  Check if the file already exists in the folder
3:   found  $\leftarrow$  false
4:   for f in  $\mathcal{F}$  do
5:     (obfName, filename',  $k_f$ ,  $iv_f$ ,  $\tau_f$ )  $\leftarrow$  f
6:     if filename = filename' then // An existing file is updated
7:       found  $\leftarrow$  true
8:        $\mathcal{F} \leftarrow \mathcal{F} \setminus (\text{obfName}, \text{filename}, k_f, iv_f, \tau_f)$ 
9:       break
10:  if !found then
11:                                      $\triangleright$  Generate file metadata for the new file
12:     $k_f \leftarrow \$ \{0, 1\}^{128}$  // file key
13:     $iv_f \leftarrow \$ \{0, 1\}^{128}$  // initialization vector
14:    obfName  $\leftarrow$  uuid() // universal unique identifier
15:                                      $\triangleright$  Encrypt the file content
16:    fileContent  $\leftarrow$  fs.read(filename)
17:    [fileContent] $_{k_f}, \tau_f \leftarrow \text{AES-GCM.Enc}(k_f, \text{fileContent}, iv_f, \epsilon)$ 
18:    server.putFile(obfName, [fileContent] $_{k_f}$ )
19:                                      $\triangleright$  Update metadata
20:     $\mathcal{F} \leftarrow \mathcal{F} \cup (\text{obfName}, \text{filename}, k_f, iv_f, \tau_f)$ 
21:  return  $\mathcal{F}$ 

```

**File download.** If the file version on the server is more recent than the one on the client, the remote version is downloaded on the client executing Algorithm 8. The function `DOWNLOAD` receives as input the files' metadata  $\mathcal{F}$ , the filename `filename` and the obfuscated name `obfName`. The file metadata corresponding to the obfuscated name is retrieved from the files' metadata. If the files' metadata does not contain an entry with the specific obfuscated name an error is showed by the function `SHOWERROR` to the user and the operation is terminated. This happens if the server reports in the versioning information a file for which no file metadata is present in the folder metadata. The procedure fetches the file content from the server, decrypts it and stores it in the local file system. At the end, the set of files' metadata is returned and the sync run continues.

This concludes the overview of the E2EE module. To briefly summarize what was presented:

- access to the E2EE data is protected by a 12 word mnemonic that each user should remember and keep secret.
- Every user has a master key that is used to protect the key material used in the files encryption. The master key is stored on the server encrypted with an encryption key derived from the mnemonic.
- The key material necessary to access a folder is stored encrypted on the server to support access from multiple devices.
- When a client wants to synchronize with the server, it first fetches the encrypted folder metadata from the server, then for each file an operation is select based on the local and remote versioning information. Finally, the folder metadata is encrypted and uploaded back to the server.

---

**Algorithm 8** E2EE file download

---

**Input:** set of files' metadata,  $\mathcal{F}$ ; the filename, **filename**; the obfuscated name obfName.

**Output:** set of files' metadata  $\mathcal{F}$

```
1: procedure DOWNLOAD( $\mathcal{F}$ , filename, obfName)
2:                                      $\triangleright$  Check if the file already exists in the folder
3:   found  $\leftarrow$  false
4:   for f in  $\mathcal{F}$  do
5:     (obfName, filename',  $k_f$ ,  $iv_f$ ,  $\tau_f$ )  $\leftarrow$  f
6:     if filename = filename' then
7:       found  $\leftarrow$  true
8:       break
9:   if !found then
10:                                      $\triangleright$  Show error and terminate operation
11:     showError()
12:   else
13:                                      $\triangleright$  Decrypt the file content and save it
14:     [fileContent] $_{k_f}$   $\leftarrow$  server.getFile(obfName)
15:     fileContent  $\leftarrow$  AES-GCM.Dec( $k_f$ , ([fileContent] $_{k_f}$ ,  $\tau_f$ ),  $iv_f$ ,  $\varepsilon$ )
16:     if fileContent =  $\perp$  then
17:       showError()
18:     else
19:       fs.write(filename, fileContent)
20:   return  $\mathcal{F}$ 
```

---

---

## Discovered vulnerabilities

---

In the following chapter, we describe the vulnerabilities found in Nextcloud’s end-to-end encryption module. Each presented attack will contain the prerequisites to carry out the attack, a description of how the attack works, and its consequences.

### 5.1 Metadata key insertion

**Threat model.** The attack can be carried out by anyone who has write and read access to the data folder of the server. Any malicious service provider can therefore carry out the attack.

**Attack description.** Metadata keys are encrypted using the user’s master key with RSA-OAEP. While providing confidentiality and integrity, RSA-OAEP does not provide authentication. Consequently, anyone who has access to the user’s public key can generate a valid encryption of a metadata key. This is especially problematic because, on every sync between client and server, all files’ metadata are re-encrypted by the client using the latest metadata key associated with the highest index. The lack of authenticity allows a malicious server to induce the client into using a metadata key known to the server. The following pseudocode shows how a malicious server can add a known metadata key to the folder metadata.

---

**Algorithm 9** Insert metadata key

---

**Input:** client’s public key,  $pk$ ; set of encrypted metadata keys,  $\mathcal{K}_e$ ; chosen metadata key,  $k_{md}$ .

**Output:** set of encrypted metadata keys containing the encryption of  $k_{md}$ ,  $\mathcal{K}_e$ .

```

1: procedure ADDMETADATAKEY( $pk, \mathcal{K}_e, k_{md}$ )
2:    $[k_{md}]_{pk} \leftarrow \text{RSA.Enc}(pk, k_{md})$ 
3:    $index \leftarrow |\mathcal{K}_e| + 1$ 
4:    $\mathcal{K}_e \leftarrow \mathcal{K}_e \cup \{(index, [k_{md}]_{pk})\}$ 
5:   return  $\mathcal{K}_e$ 

```

---

After the server modifies the folder metadata, any synchronization will cause the files’ metadata to be encrypted with the rogue metadata key.

We briefly describe a variant attack that does not rely on the clients supporting multiple metadata keys. The attack variant works as follows: when an E2EE folder is created, and the client uploads the folder metadata for the first time, the adversary overwrites the legitimate metadata key with a chosen metadata key. Every time a new file is added to

the folder, the client fetches the folder metadata, retrieves the rogue metadata key and uses it for the encryption of the newly generated file metadata. Note that because the overwriting happens when the folder is still empty, no file metadata is encrypted with the legitimate metadata key.

**Consequences.** The key insertion attack allows the adversary to choose the metadata key used to protect the file metadata and consequently to decrypt the file keys which are protected by the metadata key. This in turn allows the adversary to decrypt all files in the affected folder. Moreover, the adversary can also add files using the rogue metadata key to encrypt the newly created metadata file.

The attack variant is weaker than the original attack because it only allows an adversary to gain access to newly created folders. However, it was presented to show that the system would be vulnerable even if clients supported a single metadata key for every folder.

## 5.2 Empty metadata keys

**Threat model.** The attack can be carried out by anyone with write and read access to the data folder of the server. Any malicious service provider can therefore carry out such attack.

**Attack description.** This attack exploits an implementation bug. Specifically, when the client processes the folder metadata retrieved from the server, no error is generated if the metadata does not contain any metadata keys. As a consequence, when the client tries to encrypt a file metadata, the metadata key used will be  $\{0\}^{128}$ . This is because when accessing the empty map of metadata keys, a pointer to a section of memory containing all zeros is returned and cast to a char pointer. An adversary can remove all the metadata keys from the folder metadata and trigger this bug.

The following paragraph gives more details about the objects used in the code and their expected behavior. The object used to store metadata keys is of type QMap [5]. The documentation states that when a map is accessed at an index not present in the map, a default constructed value is inserted in the map and returned [6]. The metadata keys contained in the map are QByteArray objects, and the default value is a null byte array [4]. In essence, a null byte array is a pointer to a section of memory containing all zeros. To exploit this bug the adversary returns the following folder metadata

$$\begin{aligned}\mathcal{K}_e &= \{\} \\ \mathcal{F}_e &= \{(\text{obfName} = \text{dummy}, \varepsilon, \text{index} = 1, \varepsilon, \varepsilon)\}\end{aligned}$$

on the first time the client fetches the metadata. The malicious folder metadata consists of an empty set of metadata keys  $\mathcal{K}_e$  and a set of files' metadata  $\mathcal{F}_e$  containing the file metadata of a dummy file. The dummy file is included to ensure that the map (originally empty) of metadata keys is accessed at index 1 during metadata decryption 11. As explained previously, this will insert a null byte array at index zero. During metadata encryption, the client will use this value as a key to encrypt the files' metadata.

In conclusion, if a client receives a folder metadata with no metadata keys, all the file metadata will be encrypted with the key  $\{0\}^{128}$ .

**Consequences.** This attack allows the adversary to trick the client into using  $\{0\}^{128}$  as the metadata key. Similarly to the key overwriting attack, this attack allows an adversary to gain access to the contents of newly created folders. The adversary can use the all zeros

key to decrypt files metadata and recover file keys. This in turn allows the adversary to decrypt all the files added to the affected folder. Moreover, the adversary can also add files and encrypt the corresponding file metadata with the all zeros key.

### 5.3 IV reuse in file update

**Threat model.** The attack can be carried out by anyone who has read access to the data folder of the server. Any malicious service provider can therefore carry out the attack.

**Attack description.** As we can see from the file upload procedure 7, when uploading a file to an E2EE folder, the IV is freshly generated only if the file is new. Consequently, when a file is updated the same IV is reused. We note here that, according to the whitepaper, IVs and file keys should be resampled randomly on every upload for new and modified files. Therefore, the attack exploits an implementation error.

Files are encrypted using AES-GCM. AES-GCM is an AEAD scheme obtained combining AES-CTR with a Carter-Wegman MAC. The following describes how an IV reuse in AES-GCM can lead to a complete loss of confidentiality and authenticity.

*Plaintext recovery.* As we can see from figure 5.3, AES-GCM is similar to a one-time-pad where the key stream used to mask the plaintext is the encryption of the counters. If a counter is repeated under the same key, the key stream will also repeat and consequently the XOR of the ciphertext will yield the XOR of the plaintext. Hence, an IV reusage in GCM is equivalent to a two-time pad.

Let  $p_1, p_2$  and  $c_1, c_2$  be two versions of a file and their respective encryption. If the IV (and therefore the counters) is repeated then

$$p_1 \oplus p_2 = c_1 \oplus c_2.$$

The loss of security should be clear, as in  $c_1$  and  $c_2$ , the plaintexts were masked by a key stream that can be considered uniformly random distributed, while, due to the IV reuse, each plaintext is masked by another plaintext which has drastically less entropy than the key stream. Moreover, suppose  $p_1$  is known, then recovering  $p_2$  is trivial.

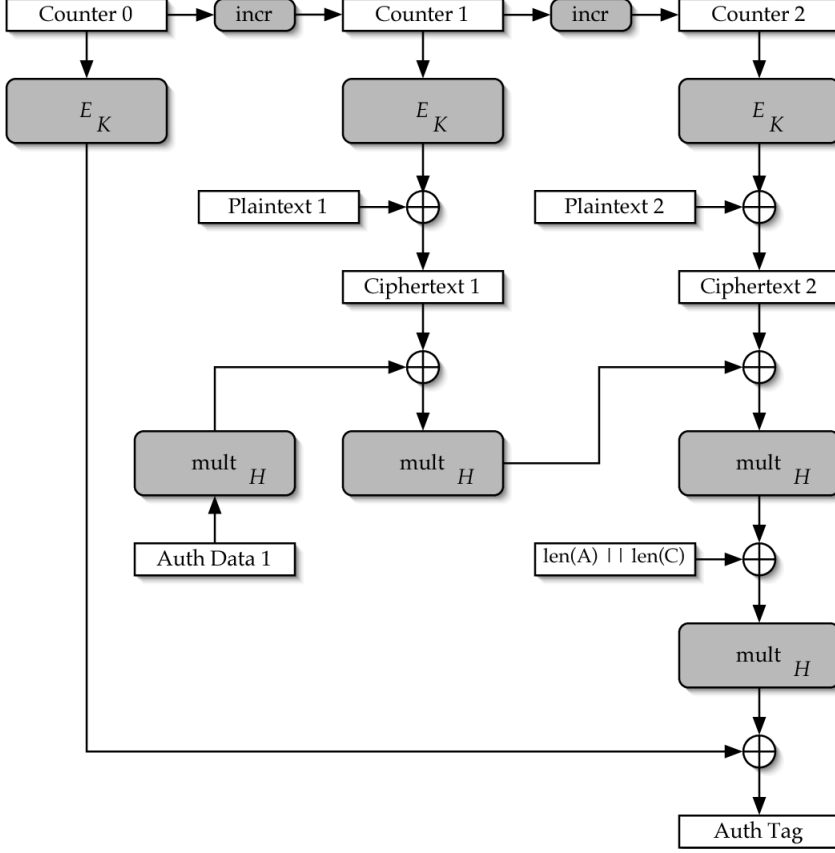
The problem of recovering  $p_1$  and  $p_2$  given  $p_1 \oplus p_2$  is well studied in the literature. In [13] the authors show how to build a language model that can separate  $p_1$  and  $p_2$  if the language of the underlying plaintext is known.

For a proof of concept, we considered a simplified case in which an end-to-end encrypted text file is modified by adding or removing a single character. Let  $d$  be the deleted or added character, and  $pos$  the position of the modification in the text. Suppose without loss of generality that  $|c_1| < |c_2|$ . Note that the position of the first modified character corresponds to the first byte in  $p_1 \oplus p_2$  different from 0. The following equations allow to recover the entire plaintext from the modification onwards

$$\begin{aligned} p_1[pos] &= d \\ p_1[i] &= c_1[i] \oplus c_2[i] \oplus p_1[i-1] \quad \forall i \text{ in } pos, pos+1, \dots, |c_2|_8. \end{aligned}$$

The implemented proof of concept iterates over the printable characters and checks, using a language detection library [26], if the recovered plaintext contains language. If the guess for  $d$  is incorrect, with high probability the recovered plaintext will contain garbled characters.

**Figure 5.1:** The authenticated encryption operation. For simplicity, a case with only a single block of additional authenticated data (labeled Auth Data 1) and two blocks of plaintext is shown. Here  $E_K$  denotes the block cipher encryption using the key  $K$ ,  $\text{mult}_H$  denotes multiplication in  $\text{GF}(2^{128})$  by the hash key  $H$ , and  $\text{incr}$  denotes the counter increment function [14].



*Tagging key recovery.* Recall that AES-GCM uses a Carter-Wegman MAC. We start by describing the Carter-Wegman MAC used in AES-GCM and then explain the Joux’s forbidden attack [12] which can be used to recover the tagging key if an IV is reused.

Let  $k, iv, A, C$  be respectively the AES-GCM key, the IV, the additional data and the ciphertext obtained by xoring the plaintext with the key stream as explained above. The additional data and the ciphertext are separately padded to a multiple of 128 bits and combined in a single message composed of blocks  $S_i$

$$S_i = \begin{cases} A_i & \text{for } i = 1, \dots, m-1 \\ A_m \parallel 0^{128-|A_m|_2} & \text{for } i = m \\ C_{i-m} & \text{for } i = m+1, \dots, m+n-1 \\ C_n \parallel 0^{128-|C_n|_2} & \text{for } i = m+n \\ \text{bits}(|A|_2, 64) \parallel \text{bits}(|C|_2, 64) & , \end{cases} \quad (5.1)$$

where  $m$  and  $n$  are the length in blocks of  $A$  and  $C$  and  $\text{bits}(x, y)$  encodes the integer  $x$  in a bit string of length  $y$ . The Carter-Wegman MAC is computed as

$$T = \sum_{i=1}^{m+n+1} S_i \cdot H^{m+n+1-i} + J, \quad (5.2)$$

where  $H = E_K(0^{128})$  and  $J = E_K(iv \parallel 0^{31} \parallel 1)$  and the operations are performed by embedding all the blocks in a fixed representation of the Galois field  $\text{GF}(2^{128})$ . The tag

is computed by evaluating a polynomial with coefficient  $S_i$  and constant term  $J$  at the value  $H$ . Note that all the coefficients  $S_i$  consisting of the additional data and ciphertext blocks are known to an adversary trying to forge a valid ciphertext. Let's now examine how an IV reuse allows for the recovery of  $H$  and  $J$ . Consider two ciphertexts, additional data  $C, A$  and  $C', A'$  obtained by encrypting two messages with AES-GCM using the same IV  $iv$ . Let  $S_i$  and  $S'_i$  be the encoding of  $C, A$  and  $C', A'$  as shown in (5.1). The respective tags can be computed as

$$\begin{aligned} T &= \sum_{i=1}^{m+n+1} S_i \cdot H^{m+n+1-i} + J, \\ T' &= \sum_{i=1}^{m'+n'+1} S'_i \cdot H^{m'+n'+1-i} + J. \end{aligned} \quad (5.3)$$

Note that  $J = E_k(iv \parallel 0^{31} \parallel 1)$  is the same in both equations due to the IV reuse. Summing the two equations in (5.3) and rearranging we obtain

$$\sum_{i=1}^{m+n+1} S_i \cdot H^{m+n+1-i} + \sum_{i=1}^{m'+n'+1} S'_i \cdot H^{m'+n'+1-i} + (T + T') = 0. \quad (5.4)$$

The coefficients of this polynomial can be computed by an adversary from the ciphertexts, the additional data and the tags. Equation (5.4) shows that  $H$  is a root of the constructed polynomial. Recovering the roots of the polynomial can be done efficiently and allows the recovery of  $H$ . Substituting  $H$  in (5.2) and solving for the constant term allows the recovery of  $J$ . The knowledge of  $H$  and  $J$  allows an attacker to create valid tags for arbitrary ciphertexts using equation (5.2).

*Framing attack to replace or modify a file.* We have seen in the previous paragraph how an IV reuse allows the recovery of plaintext and tagging key. We now show how the combination of the two attacks allows creating valid ciphertexts that decrypt to controlled plaintexts. Let  $P$  be the recovered plaintext and  $C$  the corresponding ciphertext, the key stream output used to mask the plaintext can be computed as  $P \oplus C$ . An adversary can now compute a ciphertext  $C'$  that decrypts to a desired plaintext  $P'$  as

$$C' = C \oplus P \oplus P'. \quad (5.5)$$

Since the tagging key can also be recovered, the attacker can tag the new  $C'$  creating a valid ciphertext that would decrypt correctly on the client.

**Consequences.** The attacks described in this section undermine the confidentiality and the authenticity of the end-to-end encrypted files. As shown previously, with some assumption on the underlying plaintext, full plaintext recovery is possible. Note that the assumption made can be relaxed by adopting more complex techniques. For example, in [13] the authors show that knowing the language of the underlying plaintext is sufficient to recover the plaintexts. The authors use a language model to map string of texts to the probability of that string being part of the text. The two plaintexts are recovered iteratively extending the recovered plaintext with letters or words that maximize the probability of the plaintext while also respecting the constraint that the xor of the plaintexts must equal the xor of the ciphertext. These more sophisticated techniques were not explored here because recovering the XOR of two plaintexts should already show the gravity of reusing an IV in AES-GCM. A malicious server can use the recovered key stream along with the tagging key to compute valid ciphertext that decrypts to a known controlled plaintext. This allows a malicious service provider to modify the files present in the E2EE folder.





---

# Mitigation of Attacks on Nextcloud

---

This section aims at providing some first suggestions on how the presented vulnerabilities can be mitigated.

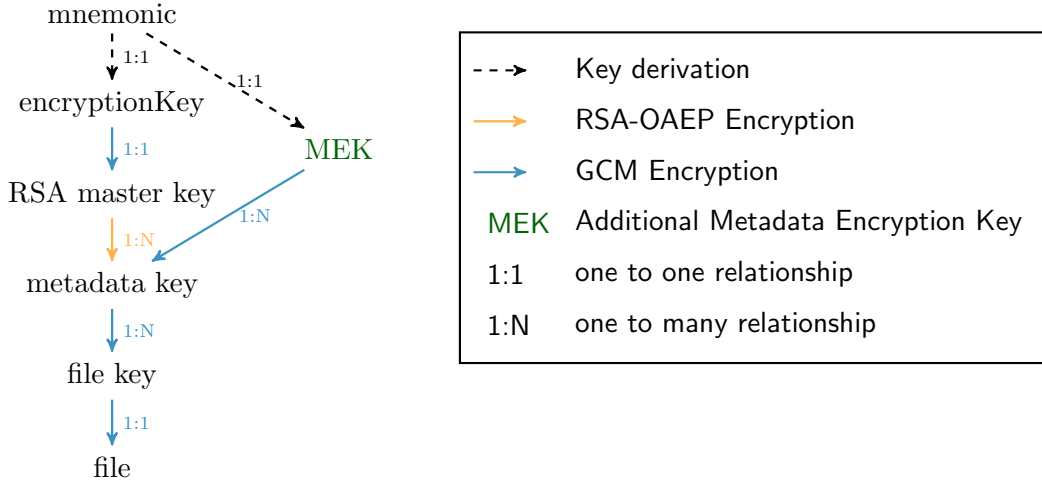
**Metadata key insertion.** As mentioned in the attack summary, the attack exploits a fundamental flaw in the cryptographic design of the E2EE module, namely that metadata keys are not authenticated.

Metadata keys must be authenticated, or else a malicious server can pick the metadata key used by the client, effectively gaining full access to the end-to-end folder. The current design encrypts metadata keys with RSA-OAEP, which provides integrity and confidentiality, but it does not provide authenticity. From our understanding, the use of asymmetric cryptography was introduced to allow a folder sharing feature. Since this feature has not yet been implemented, a simple mitigation is to drop the sharing feature and prevent the presented attack. To maintain the sharing functionality described in the whitepaper, a major revision of the E2EE module would be required.

*Simple mitigation.* If the sharing feature is removed, asymmetric encryption can be avoided and metadata keys can be protected using symmetric encryption schemes that provide confidentiality and authenticity. For example, an additional symmetric key, the metadata encryption key (MEK) is derived from the mnemonic using a secure key derivation function and an appropriate context string. The MEK is then used to encrypt the metadata keys with an AEAD scheme such as AES-GCM. The nonce used for the AEAD encryption should be sampled randomly before every encryption to avoid reuse. An AEAD scheme provides both confidentiality and integrity, preventing a malicious server from learning the metadata keys and from tampering with those keys. Since only the holder of the mnemonic can derive the appropriate MEK and use it to create properly encrypted metadata keys, we also gain an assurance concerning the authenticity of those keys. To introduce this change, the metadata encryption would need to be modified to always encrypt metadata keys with the chosen AEAD scheme. For all existing E2EE folders, a full re-encryption of the files under new file keys and metadata keys is necessary to ensure that no key material which might have leaked (due to the discovered vulnerability) is used to secure metadata or files. The modified key hierarchy is shown in figure 6.1.

We note that when implementing changes that are backward compatible, it is important to ensure that downgrade attacks are not possible. A malicious server trying to downgrade the patched version of E2EE to the original vulnerable version could always return the metadata folder with the metadata keys encrypted with RSA-OAEP. As long as clients always encrypt the metadata using symmetric encryption, the server would not be able to

**Figure 6.1:** Nextcloud's key hierarchy if the simple mitigation is applied. The MEK is added in parallel to the encryptionKey and the RSA master key.



get access to the newly added data. Each client can save a flag to memorize that a folder has migrated to the new implementation. A client receiving the legacy folder metadata after migrating to the patched one should show an error and stop.

This mitigation may seem like a reasonable and simple patch, but removes the folder sharing capability. If folder sharing is a feature that Nextcloud wants to introduce at some point in the future, this first mitigation would have to be discarded in favor of a major redesign of the E2EE system.

**Major modification.** Currently, the E2EE design shows a fundamental flaw that endangers users' data and prevents a secure implementation of the sharing feature. We believe that a redesign of the current E2EE module is necessary before moving forward with the implementation.

Following is a plausible modification to mitigate the presented vulnerability and maintain the sharing capability. A signcryption scheme can be used to protect metadata keys. The confidentiality and authenticity provided by a secure signcryption scheme ensure that a malicious server cannot recover the metadata key, nor modify it. Each user with access to a shared folder should check who generated the metadata key and act accordingly.

**Empty metadata keys.** This vulnerability relies on the fact that the map used by the client to store metadata keys allocates a default metadata key of all zeros if accessed at an index that was not already in the map. The attack can be easily prevented. Before accessing the map, a check should verify that the map contains an item at the specific index. More in general, when developing end-to-end encryption systems, the inputs provided by the server should not be trusted and therefore always checked. If the client treated the folder metadata received by the server as an untrusted input, it would check that at least one metadata key is present and abort the operation otherwise.

**IV reuse in file update.** The attack that exploits the IV reuse can be easily prevented by ALWAYS randomly sampling the IV used for file encryption, for both new and modified files as suggested in the whitepaper.

---

# Suggestions

---

This section proposes three improvements to Nextcloud's cryptographic design. In particular, while no attacks were found against the authentication process, a simpler and more efficient approach would achieve the same properties as the current one. A minor change to the E2EE setup algorithm is proposed to provide authentication to the client's public key. Finally, after briefly presenting the sharing feature described in the whitepaper, we point out an issue with the current design.

### 7.1 Improvements to authentication

The authentication process used by Nextcloud was described in section 4.1. From Nextcloud's website [21] and their security whitepaper [17], the following is required from the authentication procedure:

- A snapshot adversary with access to a snapshot of the server storage and database can only recover users' passwords by brute forcing them one at the time.
- Users can revoke access on a per-client basis from the web.
- All user's tokens must be invalidated in case the user's password is renewed.

A simpler and more efficient way to handle tokens is described in Figure 7.1. The procedures show a simplified way of creating and validating token that do not require the complicated and costly encryption and decryption mechanism used in 4.1. The table `oc_authtoken`, used to store the tokens should be modified to accommodate for the different fields that need to be stored. As in the current implementation, the server stores for each user the hash of its password in the table `oc_users`. Furthermore, all tokens are associated with a client identified `cid` which consists of the user identifier `uid` and a client name picked by the client based on the host machine it is running on. The proposed modification satisfies the requirements for the authentication procedure:

- The hashed tokens would not help a snapshot adversary to recover users' password. In fact, the stored `tokenHash` only depends on the password hash and on the token. The former is already available to a snapshot adversary in the table `oc_users` and is derived using memory hard hash functions. The latter is a high entropy value which is not vulnerable to dictionary attack.
- Per token revocation would not change from the original version.
- If a password is modified, so will the relative hash and all the tokens would become invalid.

**Figure 7.1:** Functions to create and verify tokens.

```

1: procedure CREATETOKEN(cid)
2:   token  $\leftarrow$   $\{0, 1\}^{128}$ 
3:   hashPwd  $\leftarrow$  db.get(cid, oc_users)
4:   hashToken  $\leftarrow$  new SHA512(token || hashPwd)
5:   db.put(cid, hashToken, oc_authToken)
6:   return token

1: procedure VERIFYTOKEN(token, cid)
2:   hashPwd  $\leftarrow$  db.get(cid, oc_users)
3:   storeHashToken  $\leftarrow$  db.get(cid, oc_authToken)
4:   if storeHashToken =  $\perp$  then
5:     return  $\perp$ 
6:   hashToken  $\leftarrow$  new SHA512(token || hashPwd)
7:   return hashToken == storeHashToken

```

The proposed authentication avoids asymmetric cryptography and greatly simplifies the creation and verification of the tokens. Moreover, the storage required per token on the server is reduced from a 2048-bit RSA key, to the output of a hash function. Finally, there is no need to use a password-hashing function because tokens have high entropy, and they are not vulnerable to dictionary attacks.

## 7.2 RSA master key verification

Section 4.2 describes the algorithms used by Nextcloud in the E2EE module. When a client fails to retrieve its public and private key from the keychain, as described in Algorithm 2, it fetches them from the server. While the secret key  $\mathbf{sk}$  is encrypted with AES-128-GCM under a key derived from the user's mnemonic, the public key is not integrity-protected. After decrypting the private key with the mnemonic, the client validates the key pair by executing the check in Algorithm 10.

---

**Algorithm 10** User key pair verification

---

**Input:** the user's public key,  $\mathbf{pk}$  retrieved from the server; user's secret key,  $\mathbf{sk}$ .  
**Output:** boolean indicating the validity of the key pair  $\mathbf{pk}$ ,  $\mathbf{sk}$ .

---

```

1: procedure VERIFY(pk)
2:   (e, N)  $\leftarrow$  pk
3:   (d, Nsk)  $\leftarrow$  sk
4:   r  $\leftarrow$   $\{0, 1\}^{512}$ 
5:   g  $\leftarrow$  OAEP.Encode(r)
6:   c  $\leftarrow$  ge mod N
7:   g'  $\leftarrow$  cd mod Nsk
8:   r'  $\leftarrow$  OAEP.Decode(g')
9:   return r' == r

```

---

The goal of the VERIFY procedure is to ensure that the public key actually corresponds to the authenticated private key. This procedure relies on a non-standard property of RSA-OAEP, namely it should be hard to come up with a public key such that the decryption of an encrypted random message succeeds.

As shown in the appendix A.1, a slight modification to the VERIFY would have made the system vulnerable to a key recovery attack. The intuition behind the attack is that, if the public key is not integrity-protected, an adversary could learn information about the private key by observing whether the verification fails or not. While key recovery attack proposed in A.1 does NOT apply to Nextcloud's specific instance, the integrity of the public key should rely on standard properties of the used primitives rather than non-standard ones which may not hold in general or break in subtle ways.

Following is a natural way of authenticating the public key that can substitute the current verification procedure. The client's public key should be included as additional data in the GCM encryption of the secret key. The authenticity of the public key would then rely on the integrity property provided by GCM. The client would be able to authenticate the public key because the key used to encrypt the client's RSA key is derived from the mnemonic and only known to the client.

### 7.3 Folder sharing

File sharing has not yet been implemented and the whitepaper only provides a high-level description of it. We give a summary of the sharing section of the whitepaper and later discuss the main issues.

The current design uses the server as the root of trust of a PKI that binds users' with their public keys. When user A wants to share a folder with user B, it fetches the certificate of user B from the server, validates it using the server's public key, and encrypts the latest metadata key with user's B public key. The whitepaper also describes how user A could remove user B from the shared folder. User A should generate a new metadata key and encrypt it with the public key of all the users who have access to the folder except user B. Unfortunately, we cannot give any more details because the whitepaper itself is rather vague.

The PKI approach with the server as a root of trust has limited advantages in an adversarial setting where the server can be malicious. When user A asks the server for the public key of user B, it can generate a certificate binding user B to a chosen public key (corresponding to a private key known to the server) and return it to user A. As a result, the client of user A will encrypt the metadata key to the server and not to user B. While the key distribution can still happen through the server, users should compare the fingerprint of their public keys out-of-band to bind a public key with a specific user.



---

# Conclusions

---

This research analyzed the end-to-end encryption module developed by Nextcloud. We provided a detailed description of the algorithms used to authenticate users and to secure the E2EE data. We found three vulnerabilities all leading to devastating attacks that completely shatter the confidentiality and integrity of E2EE files. For each vulnerability, we proposed mitigations that can be implemented to prevent our attacks. These results will help Nextcloud improve the security of their E2EE system, ultimately increasing the security of millions of users.

Helping Nextcloud is a worthwhile achievement. However, we believe the value of our work lies in the lessons that can be extrapolated from the found vulnerabilities. Following are three high-level takeaways that may be used proactively to prevent similar vulnerabilities in future designs and implementations.

1. **Good primitives don't imply secure systems.** While the mantra “don't roll your own crypto” starts to be widely accepted, we see more and more systems that use good algorithms and standard highly audited implementations. This may lead to a false sense of security if the chosen primitives are not fit for the requirements of the application or if they are misused. For example, the presented key insertion attack exploits the mismatch between what the designer wanted from the primitive: authentication and confidentiality, and what the primitive offered: integrity and confidentiality. Designers must understand the security guarantees offered by a primitive. Similarly, developers must know under what assumption the security guarantees hold. This would have prevented the IV reuse in file encryption.
2. **The threat model is as important for the designers as for the developers.** While the empty metadata key attack relies on an odd behavior of the object used to store metadata keys, the root cause of this vulnerability is deeper. When developing end-to-end encryption applications, the server inputs should not be trusted by the client. Had the server inputs been sanitized, the presented attack would not have been possible. In a threat model where the server is adversarial, developers must distrust all the server actions. This mindset is especially difficult because it requires developers to distrust code that they may have developed themselves.
3. **Design→Release→Break→Patch should not be an acceptable model.** Firstly, systems created with this reactive model, if trusted, may increase the perceived level of security and endanger users' privacy. Systems should be assumed to be insecure unless proven otherwise. A fully formal model and security reduction would need to be produced, specifying the protocol in full, along with the adversarial model and the computational assumptions under which the proposed protocol is secure. While this is a considerable task, the history of cryptography “in the wild” has shown that

assurances below this level do not suffice. Secondly, rolling out bad design choices is complicated and often leads to unsatisfying patches. This process is especially hard in protocols, such as E2EE cloud storage, where the involved parties maintain state.

**Future Work.** The proposed mitigations, especially the one mitigating the key insertion attack, were ideas on how to improve the current design of Nextcloud. An interesting and useful continuation of our work would try to prove that the patched system does indeed satisfy the required security guarantees.

Looking at the current internet, the cryptographic community has managed to prove and standardize secure protocols for client-server communication (TLS1.3 [10]) and E2EE messaging (Signal [3]). The ultimate goal should be to reach a similar state for E2EE cloud storage. The path toward a secure protocol for E2EE cloud storage is long but necessary. Our work on Nextcloud and Haller's[11] on Mega showed that, without a formal security model and a standardized protocol, well-established companies struggle to provide users with the necessary privacy guarantees. In conclusion, future work should go in the direction of formalizing a security model and designing a secure protocol for E2EE cloud storage. This is an ambitious and complicated work, but the result would greatly improve the current level of users' privacy.



## Appendix A

---

# Appendix

---

### A.1 Key recovery attack

This attack was inspired by the procedure used in Nextcloud to verify that a private key corresponds to a particular public key. It is important to notice that the developed attack does NOT apply to Nextcloud because the RSA decryption process gets the modulus from the private key which is integrity-protected.

**Attack scenario.** While developing the attack we considered the scenario in which a malicious service provider tries to learn the secret key of one of its clients. The secret key is stored encrypted and integrity-protected on the server, while the public key is stored in plaintext. The adversary can interact with the client providing fake public keys and learn if the verification process succeeds. We model this with the oracle `VERIFY` in Algorithm 11. For reference, the verify procedure used by Nextcloud is also reported on the right under the name `VERIFYNEXTCLOUD`. The main difference, marked in gray, is that the modulus used in the decryption in line 8 comes from the secret key `sk`. This oracle can arise in

---

#### Algorithm 11 User key pair verification

---

On the left, the oracle considered for the attack. On the right, the verify procedure used by Nextcloud.

**Input:** the user's public key, `pk` retrieved from the server; user's secret key, `sk`.

**Output:** boolean indicating the validity of the key pair `pk`, `sk`.

1: <b>procedure</b> <code>VERIFY(pk)</code>	1: <b>procedure</b> <code>VERIFYNEXTCLOUD(pk)</code>
2:   // Oracle necessary for the attack	2:   // Nextcloud verify procedure
3: $(e, N) \leftarrow pk$	3: $(e, N) \leftarrow pk$
4: $d \leftarrow sk$	4: $(d, N_{sk}) \leftarrow sk$
5: $r \leftarrow \$ \{0, 1\}^{512}$	5: $r \leftarrow \$ \{0, 1\}^{512}$
6: $g \leftarrow \$ \text{OAEP.Encode}(r)$	6: $g \leftarrow \$ \text{OAEP.Encode}(r)$
7: $c \leftarrow g^e \bmod N$	7: $c \leftarrow g^e \bmod N$
8: $g' \leftarrow c^d \bmod N$	8: $g' \leftarrow c^d \bmod N_{sk}$
9: $r' \leftarrow \text{OAEP.Decode}(g')$	9: $r' \leftarrow \text{OAEP.Decode}(g')$
10: <b>return</b> $r' == r$	10: <b>return</b> $r' == r$

---

a client-server setting, in which the client fetches its public key from the server and the public key is not integrity-protected.

**Preliminaries.** In the rest of the attack description we will use two well-known functions in number theory: the Euler’s totient function  $\phi(N)$  and the Carmichael function  $\lambda(N)$ . The former counts the positive integers up to  $N$  coprime to  $N$ . The latter outputs the smallest positive integer  $m$  such that, for all elements  $a \in \mathbb{Z}_N^*$ ,  $a^m \equiv 1 \pmod{N}$ . In other words, the Carmichael function is the least common multiple of the orders of all elements in  $\mathbb{Z}_N^*$ . Let  $N = \prod_{i=1}^k p_i^{v_i}$  where  $p_1 < p_2 < \dots < p_k$  are primes and  $v_1, v_2, \dots, v_k$  are positive integers. Then  $\lambda(N)$  is the least common multiple of the  $\lambda$  of each of its prime power factors

$$\lambda(N) = \text{lcm}(\lambda(p_1^{v_1}), \lambda(p_2^{v_2}), \dots, \lambda(p_k^{v_k})).$$

By Carmichael’s theorem, the Carmichael function of a prime power can be computed as

$$\lambda(p^v) = \begin{cases} \frac{1}{2}\phi(p^v) & \text{if } p = 2 \wedge v \geq 3 \\ \phi(p^v) & \text{otherwise.} \end{cases}$$

Note that in a typical RSA key generation, where the modulus  $N$  is a product of two large primes, the Carmichael and the totient function coincide. For an RSA modulus  $N$ , setting  $e$  and  $d$  such that  $ed \equiv 1 \pmod{\lambda(N)}$  is sufficient to ensure RSA correctness.

**Attack description.** The attack describes an interesting way of breaking RSA-OAEP encryption if the attacker can modify the public key  $pk$  used by a client and has access to the oracle shown in 11.

To introduce the idea behind this attack, let’s take a step back from the typical RSA scheme where the modulus  $N$  is a product of two large primes  $p$  and  $q$  and let’s consider a generic triplet  $(e', d, N')$ . For this triplet, RSA correctness holds if  $e'd \equiv 1 \pmod{\lambda(N')}$ . That is, for all possible  $g \in \mathbb{Z}_{N'}^*$ ,  $g^{ed} \equiv g \pmod{N'}$ . During RSA key generation, the value of  $e'$  is typically a fixed value and the value  $d$  is derived as the inverse modulo  $\lambda(N')$  of  $e'$ . The idea behind our attack is to reverse this perspective and consider the user’s private exponent  $d$  as fixed and look for the value  $e'$  for which correctness holds. If we can find this  $e'$  value, then we know that  $e'd \equiv 1 \pmod{\lambda(N')}$ .

On a high level, the attack proceeds as follows: the adversary overwrites the user’s  $pk = (e, N)$  with a new  $pk' = (e', N')$  and sends this to the client. The client will try to verify the authenticity of  $pk'$  using Algorithm 11. If the triplet  $(e', d, N')$  satisfies RSA correctness, then the verification succeeds, despite  $pk \neq pk'$ . From the correct verification of the fake  $pk'$ , the adversary learns that  $e'd \equiv 1 \pmod{\lambda(N')}$  since this is the necessary condition for RSA correctness to hold. Note that, if RSA correctness does not hold for the triplet  $(e', d, N')$ , the OAEP decoding step will fail with high probability.

The following paragraphs describe:

- what the adversary learns when a fake  $pk'$  verifies correctly
- how to construct one  $pk'$  to minimize the number of oracle queries
- how to leverage a valid  $pk'$  to efficiently recover the full private key
- how to deal with corner cases

*Information leaked by a successful verify query.* To understand what information is leaked by the oracle, we study under which condition a public key verifies correctly. All the variables used in this paragraph come from the oracle procedure `VERIFY`. In order for a public key  $(e', N')$  to verify correctly, the decrypted  $r'$  must be equal to  $r$ . This can happen in two cases:

- case 1:  $g' = g$

- case 2:  $g' \neq g \wedge \text{OAEP.Decode}(g) = \text{OAEP.Decode}(g')$ .

From lines 7 to 8 in the VERIFY oracle, we can see that

$$g' = g^{\text{de}} \pmod{N}. \quad (\text{A.1})$$

For case 1 to be true

$$\text{de} = 1 + k \cdot \text{ord}(g, N) \quad (\text{A.2})$$

has to hold. In fact, substituting (A.2) in equation (A.1) we obtain

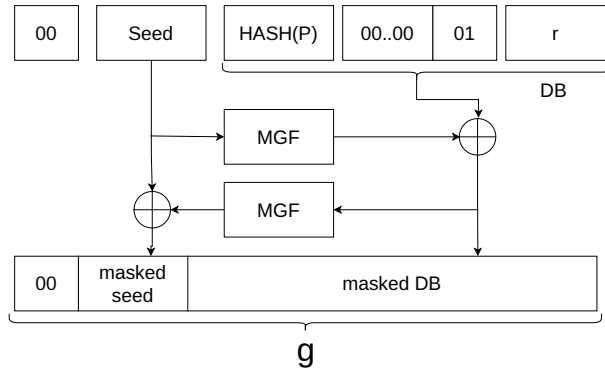
$$g' \equiv g^{1+k \cdot \text{ord}(g, N)} \equiv g \cdot g^{k \cdot \text{ord}(g, N)} \equiv g \pmod{N}. \quad (\text{A.3})$$

By definition, any element raised to a multiple of its order is equal to the identity. Moreover, the probability of case 2 being true is extremely low. Recall that the proof for RSA-OAEP IND-CCA security is in the random oracle model. If we consider the mask generation function as a random oracle, the probability of a  $g' \neq g$  verifying correctly is extremely low.

According to the RSA-OAEP standard [9], the length of the encoding  $g$  (and  $g'$ ) is equal to the size of the modulus  $N'$ . If we consider the output of the decoding of  $g'$  uniformly random, the probability of  $g' \neq g$  decoding to  $r$  is

$$\mathbb{P}[g' \neq g \wedge \text{OAEP.Decode}(g) = \text{OAEP.Decode}(g')] = \frac{2^{|\text{seed}|_2}}{2^{|N'|_2}}, \quad (\text{A.4})$$

where **seed** is the random value used in the OAEP-Encode step shown in picture A.1.



**Figure A.1:** Encoding step of RSA-OAEP.  $P$  is an octet string containing the encoding parameters.

Since we will use  $N'$  of length  $|N'|_2 \geq 1226$  and  $|\text{seed}|_2 = 256$ , the probability of case 2 being true is lower or equal to  $2^{-1194}$ . Consequently, if the VERIFY oracle returns true, equation (A.2) holds with high probability.

*How to construct  $pk'$  to minimize oracle queries.* Finding a pair  $(e', N')$  that verifies correctly and which minimizes the number of pairs which need to be tested (i.e. oracle queries) before verification succeeds is difficult for two main reasons:

1. given a modulus  $N'$ , the set of candidate values for  $e'$  is  $\mathbb{Z}_{\lambda(N')}^*$
2.  $N'$  must be at least 1226 bits long for OAEP-Encode to succeed.

An adversary can find the  $e'$  value for which  $(e', d, N')$  is a valid RSA triplet by testing all possible public keys  $(e', N')$  with  $e' \in \mathbb{Z}_{\lambda(N')}^*$  using the VERIFY oracle. This implies

that minimizing the size of  $\mathbb{Z}_{\lambda(N')}^*$  is crucial for minimizing the number of oracle queries. Finding an  $N'$  that satisfies the length constraint while keeping  $\lambda(N')$  small is not trivial. We provide a heuristic analysis of this problem and one approximated solution.

Let  $s_1 < s_2 < \dots < s_k$  be prime numbers and  $m_1, m_2, \dots, m_k$  positive integers. The following sets

$$S := \{(s_1, m_1), (s_2, m_2), \dots, (s_k, m_k)\} \quad (A.5)$$

$$P_{s,m} := \{s^i \mid 0 \leq i \leq m\} \quad (A.6)$$

$$P := \prod_{(s,m) \in S} P_{s,m} \quad (A.7)$$

$$C := \left\{ \prod_{i=1}^k x_i \mid (x_1, x_2, \dots, x_k) \in P \right\} \quad (A.8)$$

will be used in the analysis. The product of sets in (A.7) indicates the  $k$ -ary Cartesian product of all the sets  $P_{s,m}$ , where  $k$  is the size of  $S$ . Note that given a set of small primes and maximal exponents  $S$ , all the other sets are then fixed.  $C$  is a set of candidates that will be used to construct a modulus  $N'$  as described in equation (A.9).

$$N' = \prod_{c \in C} (c+1)^{\chi(c+1)} \quad (A.9)$$

In the equation the indicator function

$$\chi(y) = \begin{cases} 1 & y \text{ is prime} \\ 0 & \text{otherwise} \end{cases} \quad (A.10)$$

is used to filter for values that are prime. Since the constructed  $N'$  is of the form  $\prod_i p_i$ , for  $p_i$  prime, equation (A.11) holds by construction.

$$\begin{aligned} \lambda(N') &= \text{lcm}_i(p_i-1) \\ &= \text{lcm}_i(c_i) \\ &= \prod_{(s,m) \in S} s^m \end{aligned} \quad (A.11)$$

Note how the obtained  $\lambda$  only depends on the original set  $S$ . Moreover, with this value of  $\lambda$ , the constructed  $N'$  is the largest achievable modulus. In fact, all the possible combinations of the factors of  $\lambda$  were used to construct  $N'$ . In order to increase the modulus one would have to multiply  $\lambda$  by a new factor.

To build some intuition on how to construct an optimal set  $S$ , let's examine how the size of the set of candidates  $C$  changes if we go from a set  $S_0$  to

- $S_1 = (S_0 \setminus \{(s_i, m_i)\}) \cup \{(s_i, m_i + 1)\}$
- $S_2 = S_0 \cup \{(p, 1)\}$ .

In the first case, the maximal exponent for one of the primes in  $S_0$  is increased by one. In the second case, a new prime  $p$  is included in the set. It is easy to see that

$$\begin{aligned} |C_1| &= |C_0| + \frac{|C_0|}{m_i + 1} \\ |C_2| &= 2|C_0| \end{aligned} \quad (A.12)$$

are the sizes of the sets of candidates generated by  $S_1$  and  $S_2$ . If we assume the elements in the candidate sets to be randomly distributed we can compute the expected size of  $N'$ . The indicator function  $\chi(y)$  can be seen as a Bernoulli distributed random variable  $\chi(y) \sim \text{Ber}(\frac{1}{0.69 \log_2(y)})$ . The probability of  $\chi(y)$  evaluating to 1 comes from the prime number theorem (PNT), which describes the asymptotic distribution of the primes among positive integers. The PNT formalizes the intuitive idea that primes become less common as they grow larger.

$$\begin{aligned}
\mathbb{E}[|N'|_2] &= \mathbb{E}[\log_2(\prod_{i=1}^{|C|} (c_i + 1)^{\chi(c_i+1)})] \\
&= \sum_{i=1}^{|C|} \mathbb{E}[\chi(c_i + 1) \log_2(c_i + 1)] \\
&= \sum_{i=1}^{|C|} \frac{1}{0.69 \log_2(c_i + 1)} \log_2(c_i + 1) \\
&= 1.44|C|
\end{aligned} \tag{A.13}$$

Equations (A.13) and (A.12) show that, on average, adding a prime number to  $S$  increases the size of  $N'$  more than increasing the maximal exponent of an already used prime. However, while increasing the exponent of an existing prime  $s_i$  has a fixed cost on the total lcm of  $|s|_2$ , the cost of adding a prime number  $p$  increases with  $|p|_2$ . Moreover, the increase in the  $\mathbb{E}[|N'|_2]$  when adding a prime does not depend on the prime itself. In conclusion, in order to construct a sufficiently large modulus  $N'$  with  $\lambda(N')$  as small as possible, smaller primes should be preferred to bigger ones as they keep  $\lambda(N')$  smaller while having the same effect on  $|N'|_2$ .

*Approximated solution.* In the following paragraph an approximate solution is proposed and, based on the previous analysis, some arguments are given to why this approximated solution should be close to the optimal one. As a reference, an exhaustive search over the sets

$$\begin{aligned}
S_{\text{min\_search}} = \{ & (2, m_1), (3, m_2), (5, m_3), (7, m_4), (11, m_5), (13, m_6), \\
& (17, m_7), (19, m_8), (23, m_9), (29, m_{10}), (31, m_{11}) \}
\end{aligned} \tag{A.14}$$

with possible maximal exponents  $0 \leq m_i \leq \text{ceil}(\log_{s_i}(256))$  finds a valid  $N'$  with  $|\lambda(N')|_2 = 22$  and

$$\begin{aligned}
S^* = \{ & (2, 4), (3, 2), (5, 2), (7, 1), (11, 1), (13, 1), (17, 0), \\
& (19, 0), (23, 0), (29, 0), (31, 0) \}.
\end{aligned} \tag{A.15}$$

Limiting the powers of the small prime was necessary for the search to finish in a reasonable amount of time. While there are no profound reasons to set the bound to 256, the following paragraph shows that this limit is large enough to argue that any other candidate solution outside the considered search space would not be better than the found one. Regardless of the search being limited to the maximal exponents  $m_i$ ,  $S^*$  should be close to the best solution. Any set  $S^\sim$  leading to a smaller  $\lambda(N')$  containing the same primes but a maximal exponent greater than the ones considered in the search would have to at least set to zero the maximal exponent of a previously used prime leading to a smaller set of candidates and consequently, on average, a smaller  $N'$ . One might also consider adding new primes to  $S^*$ . Again, any prime greater than 31 would require removing a factor  $f$  of size  $|f|_2 = 6$

from the powers in  $S^*$  to have a significant reduction in the bit length of  $\lambda(N')$ . This could only be achieved by setting at least one maximal exponent to 0. Consequently, the candidate set would be smaller (same number of primes but with lower exponents) and on average result in a smaller  $N'$ .

While the previous reasoning is not a proof, it should be a convincing argumentation that the found solution is a good approximation of the optimal one.

*Full key recovery.* The full key recovery consists of two phases: in the first phase, the adversary sets to  $N_1 = N'$  generated previously (A.15) and for all  $\mathbf{e}' \in \mathbb{Z}_{\lambda(N_1)}^*$  it tests  $\text{VERIFY}(\mathbf{e}', N_1)$  until it finds the value  $\mathbf{e}' = \mathbf{e}_1$  that verifies correctly and therefore satisfies equation

$$\mathbf{e}_1 \mathbf{d} \equiv 1 \pmod{\lambda(N_1)}. \quad (\text{A.16})$$

After this first phase in which the adversary finds the correct value  $\mathbf{e}_1$  brute-forcing all the possible values, the adversary starts adaptively querying the oracle with a strategy that leaks  $\mathbf{d}$  bit by bit with few oracle queries per bit.

This paragraph describes how the knowledge of  $\mathbf{e}_1, N_1$  can be leveraged to extend our knowledge of  $\mathbf{d}$  by one bit with few oracle queries. The same strategy can be iterated until all bits of  $\mathbf{d}$  are leaked. Suppose  $N_1 = \mathbf{n}_1 2^{\mathbf{m}}$ , with  $\gcd(\mathbf{n}_1, 2) = 1$ . Note that the  $N_1$  which we used in the first phase of the attack satisfies this assumption with  $\mathbf{m} = 4$ . The following description is general for any value of  $\mathbf{m}$ , because this strategy will be iteratively repeated with increasing values of  $\mathbf{m}$  to leak  $\mathbf{d}$  bit by bit.

We have that

$$\lambda(2^{\mathbf{m}}) = \begin{cases} 1 & \text{if } \mathbf{m} = 1 \\ 2 & \text{if } \mathbf{m} = 2 \\ 2^{\mathbf{m}-2} & \text{for all } \mathbf{m} > 2. \end{cases} \quad (\text{A.17})$$

Moreover, let  $\mathbf{d}_{\mathbf{m}} = \mathbf{d} \pmod{\lambda(2^{\mathbf{m}})}$  be the value of  $\mathbf{d}$  truncated to the first  $\log_2(\lambda(2^{\mathbf{m}}))$  bits.

From (A.16) we know that

$$\mathbf{e}_1 \mathbf{d} \equiv 1 \pmod{\lambda(\mathbf{n}_1)} \quad (\text{A.18})$$

$$\mathbf{e}_1 \mathbf{d} \equiv 1 \pmod{\lambda(2^{\mathbf{m}})}. \quad (\text{A.19})$$

The idea is now to extend our knowledge of  $\mathbf{d}$  looking for a pair  $\mathbf{e}_2, N_2$ , with  $N_2 = 2N_1 = 2^{\mathbf{m}+1}\mathbf{n}_1$  that satisfies

$$\mathbf{e}_2 \mathbf{d} \equiv 1 \pmod{\lambda(N_2)}. \quad (\text{A.20})$$

Note that  $\lambda(N_1) \mid \lambda(N_2)$  and therefore

$$\mathbf{e}_2 \equiv \mathbf{e}_1 \pmod{\lambda(N_1)}. \quad (\text{A.21})$$

For  $\mathbf{e}_2, N_2$  to satisfy equation (A.20), the following two equations

$$\begin{aligned} \mathbf{e}_2 \mathbf{d} &\equiv 1 \pmod{\lambda(\mathbf{n}_1)} \\ \mathbf{e}_2 \mathbf{d} &\equiv 1 \pmod{\lambda(2^{\mathbf{m}+1})} \end{aligned} \quad (\text{A.22})$$

must hold. Note that knowing an  $\mathbf{e}_2$  that satisfies the second equation, is equivalent to knowing  $\mathbf{d}_{\mathbf{m}+1}$ .

We know that  $\mathbf{e}_1$  is a solution for the first equation. Moreover, by equation (A.19) we know that  $\mathbf{d}_{\mathbf{m}} = \mathbf{e}_1^{-1} \pmod{\lambda(2^{\mathbf{m}})}$ .

There are only two possible values for  $d_{m+1}$ , namely

$$\begin{aligned} d_{m+1}^{(0)} &= d_m \\ d_{m+1}^{(1)} &= d_m + \lambda(2^m). \end{aligned}$$

This should be intuitive as we are basically guessing one additional bit of  $d$ . We can rewrite the system of equations (A.22) as

$$\begin{aligned} e_2 &\equiv e_1 \pmod{\lambda(N_1)} \\ e_2 &\equiv (d_{m+1}^{(i)})^{-1} \pmod{\lambda(2^{m+1})}. \end{aligned} \tag{A.23}$$

Solving it for  $i = 0$  and  $i = 1$  returns two candidates  $e_2$  values  $e_2^{(0)}, e_2^{(1)}$ . Testing  $e_2^{(0)}, N_2$ , and  $e_2^{(1)}, N_2$ , against the VERIFY oracle will retrieve the unique  $e_2$  that satisfies equation (A.20) and consequently the correct guess for  $d_{m+1}$ .

Using the oracle to eliminate the incorrect  $e_2$  value requires some attention. In particular, as shown in the appendix A.2, RSA correctness fails for inputs  $g$  such that  $2 \mid \gcd(g, N_2)$  because our modulus contains powers of two. This means that the VERIFY oracle will fail with probability 0.5 if the tested  $e_2$  satisfies equation (A.20). Moreover, due to (A.21), decryption would work for all the odd encodings  $g$  such that  $\text{ord}(g, 2^{m+1}) < \lambda(2^{m+1})$ . The incorrect candidate would however fail the VERIFY oracle if  $\text{ord}(g, 2^{m+1}) = \lambda(2^{m+1})$ . These three observation show that a single oracle query is not sufficient to distinguish the correct guess from the wrong one because in both cases the oracle can succeed or fail. However, the success probability of the verify oracle is different for the two candidates, thus with enough oracle attempts we can confidently distinguish the correct  $e_2$  from the wrong one. The details of how this filtering works are shown in the appendix A.3.

Note that this process allows us to increase by one bit our knowledge of  $d$ .

Repeating this process multiple times allows us to retrieve  $d_t = d \pmod{2^t}$  for increasingly higher values of  $t$ . Once enough bits of  $d$  are known (roughly  $|d|_2/4$ ), a standard lattice attack described in [15] section 4.2.9 can be used to recover the full private key.

The whole derivation assumed knowledge of a  $e_1, N_1$  that verifies correctly. In practice, we use  $N_1 = N'$  generated previously (A.15) and for all  $e' \in \mathbb{Z}_{\lambda(N_1)}^*$  we would try  $\text{VERIFY}(e', N_1)$ . With the particular choice of  $N_1$  this would require at most  $\phi(\lambda(N_1)) = 691200$  oracle queries.

To sum up, after finding  $e_1, N_1$  that verifies correctly with a cost of approximately  $2^{20}$  oracle queries, we can start leaking  $d$  bit by bit.

*Corner cases.* The described attack fails if  $\gcd(d, \lambda(N_1)) \neq 1$  and therefore equation (A.16) has no solution. By construction the prime factors of  $\lambda(N_1)$  are 2, 3, 5, 7, 11, 13. Moreover, let  $N$  be the modulus of the legitimate public key.  $\phi(N)$  is even and therefore the private key  $d$  is necessarily odd, otherwise it would not have an inverse. Let  $E_i$  be the event that the  $i$ th factor divides  $d$ ,

$$\mathbb{P}[\gcd(d, \lambda(N_1)) = 1] = 1 - \mathbb{P}[\cup_i E_i] \sim 0.3.$$

The proposed attack can be adapted to work even if  $\gcd(d, \lambda(N_1)) \neq 1$ . In particular, if after attempting all the possible values  $e_1$  once, none of them resulted in a successful verify query we are certain that  $d$  and  $\lambda(N_1)$  have a common prime. We can create a new  $N'_1$  substituting one of the prime factors of  $\lambda(N_1)$  for a bigger one. Repeating the process until a VERIFY query succeeds ensures the attack to be successful for all private keys.

**Attack complexity.** As we have seen, after the first successful VERIFY oracle query, it is possible to leak bits of  $d$  using only few oracle queries. The proof of concept showed that, on average, less than of 65 oracle queries are sufficient to determine an additional bit of  $d$  with 0.99999 confidence. Such confidence ensures that the probability of recovering 512 bits correctly is 0.99. An RSA key of 2048 bits can be fully recovered with a lattice attack given the knowledge of 512 bits of  $d$ . We refer to section 4.2.9 in [15] for the details of the lattice attack. The most expensive part of the attack is the first phase. The cost of this phase is proportional to the size of  $\lambda(N_1)$ . We managed to create a valid  $N_1$  with  $\lambda(N_1) = 2^4 3^{25} 5^2 7^{11} 13$ . Since, in the worst case, all the values in  $\mathbb{Z}_{\lambda(N_1)}^*$  have to be verified, the attack will require  $\phi(\lambda(N_1))$  oracle queries for the first successful VERIFY and 60 queries per bit of  $d$  leaked. The total attack cost would be  $2^{20}$ . This is the cost for the basic attack that does not handle the case in which  $\lambda(N_1)$  and  $d$  are not coprime. As previously discussed,  $\lambda(N_1)$  and  $d$  are coprime with probability  $\approx 1/3$ . Therefore, the basic attack successfully recovers one in three users' private keys. If  $\lambda(N_1)$  and  $d$  are not coprime, the attack becomes more complicated and costly. Essentially, the first phase would have to be repeated with different values  $N_1$  constructed by iteratively substituting prime factors of  $\lambda(N_1)$  for bigger primes until all the one in common with  $d$  are removed.

## A.2 RSA encryption with multiple primes in the modulus

The key recovery attack described in section A.1 works by providing the client with fake public keys  $pk = (e, N)$  and observing the result of the VERIFY oracle 1. This section first shows that RSA correctness holds for a generic square free modulus which may contain more than only two primes. In the end we show that if the RSA modulus is divisible by the power of a prime (non-square free), then RSA correctness fails for some inputs.

Let's start considering a modulus  $N = \prod_{i=1}^k p_i$ , with  $k > 2$  and  $p_i$  primes. This is not a standard RSA modulus, and therefore we show that correctness of RSA

$$m^{ed} \equiv m \pmod{N} \quad (\text{A.24})$$

still holds for any message  $m < N$ .

By the Chinese remainder theorem

$$m^{ed} \equiv m \pmod{N} \iff \left\{ m^{ed} \equiv m \pmod{p_i} \quad \forall i \in 1 \dots k \right\}. \quad (\text{A.25})$$

In the following we will examine a single congruence equation at the time and therefore the subscript  $i$  is dropped for ease of notation. For each congruence equation, two cases can arise

- $\gcd(m, p) = 1$
- $\gcd(m, p) = p$ .

We assume  $d$  to be computed as in the normal RSA key generation, hence  $ed \equiv 1 \pmod{\phi(N)}$  and consequently  $ed \equiv 1 \pmod{\phi(p)}$ . In the first case the congruence equation in (A.25) can be rewritten as

$$m^{ed} \equiv m \pmod{p} \iff m^{t\phi(p)+1} \equiv m \pmod{p} \iff m \equiv m \pmod{p} \quad (\text{A.26})$$

and always holds. The last iff comes from the fact that an element raised to a multiple of its order is equivalent to the neutral element. Let's now consider the second case. Since  $\gcd(m, p) = p$ ,  $m = rp$  for some integer  $r$ . The congruence equation in (A.25) can be



rewritten as

$$\begin{aligned}
 m^{ed} &\equiv m \pmod{p} \iff \\
 (rp)^{ed} &\equiv rp \pmod{p} \iff \\
 r^{ed} p^{ed-1} &\equiv r \pmod{1} \iff \\
 0 &\equiv 0 \pmod{1}
 \end{aligned} \tag{A.27}$$

and again always holds. We have shown that each congruence equation in Equation A.25 holds and therefore RSA correctness is preserved.

We will now examine what happens if  $N$  is non-square free. This case is relevant in the second phase of the key recovery attack. During this phase the private key of the user is leaked bit by bit. This is done using moduli that contain increasing powers of 2. Suppose  $N = \prod_{i=1}^k p_i \cdot p^z$ . All the congruence equation modulo  $p_i$  will hold for the same reasoning as before. Let's now examine the congruence equation

$$m^{ed} \equiv m \pmod{p^z}. \tag{A.28}$$

If  $\gcd(m, p^z) = 1$ , or  $\gcd(m, p^z) = p^z$  we fall back in the two previously examined cases and correctness holds. We will now show that if  $\gcd(m, p^z) = p^{\bar{z}}$  for some  $\bar{z} < z$ , RSA correctness will fail. Let  $m = rp^{\bar{z}}$  for some integer  $r$ . Rewriting equation (A.28), we obtain

$$\begin{aligned}
 (rp^{\bar{z}})^{ed} &\equiv rp^{\bar{z}} \pmod{p^z} \iff \\
 r^{ed} p^{\bar{z}(ed-1)} &\equiv r \pmod{p^{z-\bar{z}}} \iff \\
 r^{ed-1} p^{\bar{z}(ed-1)} &\equiv 1 \pmod{p^{z-\bar{z}}} \iff \\
 r^{ed-1} p^{\bar{z}ed-z} p^{z-\bar{z}} &\equiv 1 \pmod{p^{z-\bar{z}}}
 \end{aligned} \tag{A.29}$$

In the derivation, we used the fact that  $\gcd(r, p^z) = 1$  and therefore  $r$  has an inverse. Moreover, in the last line we made explicit the fact that the term on the left is a multiple of  $p^{z-\bar{z}}$ , and therefore it is equivalent to zero. Thus, the last congruence does not hold. In conclusion, with non-square free modulus, for all the messages  $m$  such that  $\gcd(m, p^z) = p^{\bar{z}}$  for some  $\bar{z} < z$ , RSA correctness fails.

### A.3 Oracle queries to filter candidate public keys

This section complements the enhanced key recovery attack A.1. We will briefly recap the setting used in the key recovery attack and then show how we can use the VERIFY oracle to filter out incorrect guesses. In particular let  $N_1, e_1$ , with  $N_1 = 2^{m+2}n_1$ , be a public key that verifies correctly and let  $N_2 = 2N_1$ . Knowing that

$$\begin{aligned}
 e_1 d &\equiv 1 \pmod{\lambda(n_1)} \\
 e_1 d &\equiv 1 \pmod{2^m},
 \end{aligned} \tag{A.30}$$

The goal is to find an  $e_2$  such that

$$e_2 d \equiv 1 \pmod{\lambda(n_1)} \tag{A.31}$$

$$e_2 d \equiv 1 \pmod{2^{m+1}}. \tag{A.32}$$

Intuitively,  $e_2 \equiv e_1 \pmod{\lambda(n_1)}$  and  $e_2 \equiv e_1 \pmod{2^m}$ . For example, if (A.32) holds, it also holds for powers of 2 smaller than  $2^{m+1}$ , and we already know that  $e_1$  satisfies the equation  $\pmod{2^m}$ .

As shown in the key recovery attack, there are two candidate values  $e_2^{(0)}, e_2^{(1)}$ .

**Proposition A.1** *If  $e_2$  is such that*

$$e_2 d \equiv 1 \pmod{(\lambda(N_2))} \quad (\text{A.33})$$

*holds, then*

$$\mathbb{P}[\text{VERIFY}(N_2, e_2)] = 0.5 \quad (\text{A.34})$$

**Proof** Let  $g$  be the encoding of the randomly sampled value in `VERIFY`. Appendix A.2 shows that, if  $2 \mid g$  RSA correctness fails and so will the `VERIFY` oracle. For all the other  $g$  values, hypothesis (A.33) ensures that decryption will succeed and so will the `VERIFY` oracle. In conclusion, half of the  $g$  values will result in a failing query.  $\square$

**Lemma A.2** *For all  $k > 2$ , the number of elements of max order in  $\mathbb{Z}_{2^k}^*$  is  $2^{k-3}$ .*

**Proof** Note that for all  $k > 2$ ,  $\mathbb{Z}_{2^k}^*$  is isomorphic to the direct product of two cyclic groups of order 2 and  $2^{k-2}$ . Moreover, the number of elements of max order in a cyclic group of order  $t$  is  $\phi(t)$ . Using these properties we conclude that the number of elements in  $\mathbb{Z}_{2^k}^*$  is

$$\phi(2^{k-2}) = 2^{k-3} \quad (\text{A.35})$$

$\square$

**Proposition A.3** *If  $e_2$  is such that*

$$e_2 d \equiv 1 \pmod{(\lambda(N_1))}, \quad (\text{A.36})$$

*and*

$$e_2 d \not\equiv 1 \pmod{(\lambda(N_2))}, \quad (\text{A.37})$$

*then*

$$\mathbb{P}[\text{VERIFY}(N_2, e_2)] = 0.25 \quad (\text{A.38})$$

**Proof** Let  $g$  be the encoding of the randomly sampled value in `VERIFY`. By hypothesis (A.36) decryption will work in all the subgroups of  $\mathbb{Z}_{n_1}$ . Let's now examine how decryption works in  $\mathbb{Z}_{2^{m+3}}$  knowing, by hypothesis,

$$e_2 d \equiv 1 \pmod{2^m} \quad (\text{A.39})$$

$$e_2 d \not\equiv 1 \pmod{2^{m+1}}. \quad (\text{A.40})$$

Appendix A.2 shows that, if  $2 \mid g$  RSA correctness fails and so will the `VERIFY` oracle.

We will now consider odd values of  $g$ . If

$$\text{ord}(g, 2^{m+3}) = 2^t < \lambda(2^{m+3}) = 2^{m+1},$$

then by (A.39) decryption will work. In fact,  $e_2 d = 1 + i2^m = 1 + j2^t$  for some integers  $i, j$ , and

$$g^{e_2 d} \equiv g^{1+k2^t} \equiv g \pmod{2^{m+3}}. \quad (\text{A.41})$$

If  $g$  has max order in  $\mathbb{Z}_{2^{m+3}}^*$ , decryption will fail due to hypothesis (A.40).

The target probability can be rewritten as

$$\mathbb{P}[\text{VERIFY}(N_2, e_2)] = 1 - \mathbb{P}[g \text{ is even} \mid g \leftarrow \mathbb{Z}_{2^{m+3}}] - \mathbb{P}[\text{ord}(g, 2^{m+3}) = \lambda(2^{m+3}) \mid g \leftarrow \mathbb{Z}_{2^{m+3}}^*] \quad (\text{A.42})$$

To finish the proof we have to show that the third term equals 0.25. Applying (A.2), we obtain that the number of elements of maximal order in  $\mathbb{Z}_{2^{m+3}}^*$  is  $2^m$ . Consequently,

$$\mathbb{P}[\text{ord}(g, 2^{m+3}) = \lambda(2^{m+3}) \mid g \leftarrow \mathbb{Z}_{2^{m+3}}^*] = \frac{2^m}{\phi(2^{m+3})} = \frac{2^m}{2^{m+2}} = 0.25. \quad (\text{A.43})$$

$\square$

Let without loss of generality  $\mathbf{e}_2 = \mathbf{e}_2^{(0)}$  be the value for which

$$\mathbf{e}_2 \mathbf{d} \equiv 1 \pmod{\lambda(N_2)}$$

holds, and let  $\mathbf{e}_2^{(1)}$  the incorrect guess for  $\mathbf{e}_2$ . By proposition (A.1) and (A.3),

$$\begin{aligned} \mathbb{P}[\text{VERIFY}(N_2, \mathbf{e}_2^{(0)})] &= 0.5 \\ \mathbb{P}[\text{VERIFY}(N_2, \mathbf{e}_2^{(1)})] &= 0.25. \end{aligned} \tag{A.44}$$

The result of a single VERIFY query is not sufficient to distinguish  $\mathbf{e}_2^{(0)}$  from  $\mathbf{e}_2^{(1)}$ . Let

$$\begin{aligned} \text{VERIFY}(N_2, \mathbf{e}_2^{(0)}) &\sim \text{Ber}(0.5) \\ \text{VERIFY}(N_2, \mathbf{e}_2^{(1)}) &\sim \text{Ber}(0.25) \\ \text{VERIFY}(N_2, \mathbf{e}_2) &\sim \text{Ber}(p) \end{aligned} \tag{A.45}$$

where  $\text{Ber}(\rho)$  indicates a Bernoulli distributed random variable which takes value 1 with probability  $\rho$ .

We then use hypothesis testing to distinguish the two distributions. In particular, we make two null hypotheses:

- H0:  $p \geq 0.5$
- G0:  $p \leq 0.25$ .

In the attack we keep on querying the VERIFY oracle until either one is rejected with a certain confidence. If H0 is rejected,  $p = 0.25$  and consequently  $\mathbf{e}_2 = \mathbf{e}_2^{(1)}$ . If G0 is rejected,  $p = 0.5$  and consequently  $\mathbf{e}_2 = \mathbf{e}_2^{(0)}$ . In conclusion, this method allows us, given the two  $\mathbf{e}_2^{(1)}, \mathbf{e}_2^{(0)}$ , to determine the one satisfying equation

$$\mathbf{e}_2 \mathbf{d} \equiv 1 \pmod{(\lambda(N_2))}. \tag{A.46}$$

This concludes the Appendix section. The Appendix contains the description on a key recovery attack which was developed during the project, but that did not apply to the specific instance of Nextcloud. Such attack was still reported because it raised interesting challenges and may prove useful in other settings.



---

## Bibliography

---

- [1] Qtkeychain documentation. <https://include.org/libraries/qtkeychain.html>, Last accessed on 2022-09-30.
- [2] A. Biryukov. Rfc 9106 argon2 memory-hard function for password hashing and proof-of-work applications. <https://www.rfc-editor.org/rfc/rfc9106.html>, Last accessed on 2022-09-26.
- [3] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. Cryptology ePrint Archive, Paper 2016/1013, 2016. <https://eprint.iacr.org/2016/1013>.
- [4] The Qt Company. Qbytearray class. <https://doc.qt.io/qt-6/qbytearray.html#QByteArray>.
- [5] The Qt Company. Qmap class. <https://doc.qt.io/qt-6/qmap.html>.
- [6] The Qt Company. Qmap class, access operator. <https://doc.qt.io/qt-6/qmap.html#operator-5b-5d>.
- [7] Internet Engineering Task Force. Hmac-based extract-and-expand key derivation function (hkdf). <https://www.rfc-editor.org/rfc/rfc5869>.
- [8] Internet Engineering Task Force. The oauth 2.0 authorization framework. <https://www.rfc-editor.org/rfc/rfc6749>.
- [9] Internet Engineering Task Force. Rsa cryptography specifications. <https://www.ietf.org/rfc/rfc2437.txt>.
- [10] Internet Engineering Task Force. The transport layer security (tls) protocol version 1.3. <https://www.rfc-editor.org/rfc/rfc8446>.
- [11] Miro Haller. Cloud storage systems: From bad practice to practical attacks. Master's thesis, ETH Zurich, 2022.
- [12] Antoine Joux. Authentication failures in nist version of gcm. *NIST Comment*, page 3, 2006.
- [13] Joshua Mason, Kathryn Watkins, Jason Eisner, and Adam Stubblefield. A natural language approach to automated cryptanalysis of two-time pads. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS '06*, page 235–244, New York, NY, USA, 2006. Association for Computing Machinery.

- [14] David A. McGrew and John Viega. The galois/counter mode of operation (gcm). 2005.
- [15] Gabrielle De Micheli and Nadia Heninger. Recovering cryptographic keys from partial information, by example. Cryptology ePrint Archive, Paper 2020/1506, 2020. <https://eprint.iacr.org/2020/1506>.
- [16] Steve Morgan. Data attack surface report. cybersecurity ventures, 2020.
- [17] Nextcloud. Security and authentication. <https://nextcloud.com/blog/whitepapers/security/>, Last accessed on 2022-11-30.
- [18] Nextcloud. German federal administration relies on nextcloud as a secure file exchange solution, 2018. <https://nextcloud.com/blog/german-federal-administration-relies-on-nextcloud-as-a-secure-file-exchange-solution/>.
- [19] Nextcloud. Nextcloud end-to-end encryption rfc, 2018. [https://github.com/nextcloud/end\\_to\\_end\\_encryption\\_rfc/blob/master/RFC.md#security-properties](https://github.com/nextcloud/end_to_end_encryption_rfc/blob/master/RFC.md#security-properties), Last accessed on 2022-09-12.
- [20] Nextcloud. Nextcloud grew customer base 7x, added over 6.6 million lines of code and doubled its team in 2017, 2018. <https://nextcloud.com/blog/nextcloud-grew-customer-base-7x-added-over-6-6-million-lines-of-code-and-doubled-its-team-in-2017/>.
- [21] Nextcloud. Login flow, 2022. [https://docs.nextcloud.com/server/latest/developer\\_manual/client\\_apis/LoginFlow/index.html](https://docs.nextcloud.com/server/latest/developer_manual/client_apis/LoginFlow/index.html), Last accessed on 2022-11-30.
- [22] Nextcloud. Nextcloud about page, 2022. <https://nextcloud.com/about>.
- [23] Nextcloud. Nextcloud ceo kicks off nextcloud conference with keynote speech, 2022. <https://nextcloud.com/blog/nextcloud-ceo-kicks-off-nextcloud-conference-with-keynote-speech/>.
- [24] Nextcloud. Nextcloud home page, 2022. <https://nextcloud.com/>.
- [25] Nextcloud. Nextcloud threat model, 2022. <https://nextcloud.com/security/threat-model/>, Last accessed on 2022-09-12.
- [26] Jeroen Ooms. *cld2: Google's Compact Language Detector 2*, 2022. <https://docs.ropensci.org/cld2/> (docs) <https://github.com/ropensci/cld2> (devel) <https://github.com/cld2owners/cld2> (upstream).
- [27] Niels Provos and David Mazieres. A future-adaptable password scheme. 03 2001.