End-to-End Encrypted Cloud Storage in the Wild: A Broken Ecosystem

Jonas Hofmann*
ETH Zurich
Switzerland
jonas.hofmann1@tu-darmstadt.de

Kien Tuong Truong
ETH Zurich
Switzerland
kientuong.truong@inf.ethz.ch

ABSTRACT

End-to-end encrypted cloud storage offers a way for individuals and organisations to delegate their storage needs to a third-party, while keeping control of their data using cryptographic techniques. We conduct a cryptographic analysis of various products in the ecosystem, showing that many providers fail to provide an adequate level of security. In particular, we provide an in-depth analysis of five end-to-end encrypted cloud storage systems, namely Sync, pCloud, Icedrive, Seafile, and Tresorit, in the setting of a malicious server. These companies cumulatively have over 22 million users and are major providers in the field. We unveil severe cryptographic vulnerabilities in four of them. Our attacks invalidate the marketing claims made by the providers of these systems, showing that a malicious server can, in some cases, inject files in the encrypted storage of users, tamper with file data, and even gain direct access to the content of the files. Many of our attacks affect multiple providers in the same way, revealing common failure patterns in independent cryptographic designs. We conclude by discussing the significance of these patterns beyond the security of the specific providers.

CCS CONCEPTS

• **Security and privacy** → **Cryptanalysis and other attacks**; *Key management*.

KEYWORDS

End-to-end encryption, Cloud storage, Cryptanalysis, Cryptographic protocols

ACM Reference Format:
Jonas Hofmann and Kien Tuong Truong. 2024. End-to-End Encrypted Cloud Storage in the Wild: A Broken Ecosystem. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3658644.3690309

*Also with Technische Universität Darmstadt.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690309

1 INTRODUCTION

Cloud storage is a method for offloading digital data to a remote third-party, called the *provider*, who stores and manages the data on their own infrastructure. Thanks to the low-cost and high-availability guarantees, providers such as Google Drive, iCloud, Microsoft OneDrive, DropBox or MEGA have become very popular, amassing more than four billion users combined and user data in the order of exabytes [8, 14–16, 38, 39]. Almost all of these providers use encryption-at-rest to protect the user data from external attackers. If we, however, consider the case in which the provider itself is compromised, encryption-at-rest does not provide any protection since the provider controls the encryption keys.

To secure user data in this setting, which we call the *compromised*, or *malicious, server setting*, providers have started deploying end-to-end encryption (E2EE), where user files are encrypted with keys managed by the user, rather than the server. Examples of such providers are MEGA, Nextcloud, Sync [49], Tresorit [1], Seafile [35], Icedrive [31], and pCloud [43]. Despite holding a minority of the user base in the cloud storage space, these providers are a primary choice for relevant organisations and privacy-savvy users who wish to keep control of their data. For example, Nextcloud is used by the German and Serbian governments as well as Amnesty International. Sync is used by the Canadian government, the Canadian Red Cross, and the states of Vermont and North Dakota. Tresorit lists SAP, Pfizer, and Allianz as customers.

With many important entities using E2EE cloud storage, analysing their security becomes a pressing concern. Recent works have uncovered vulnerabilities in two of the most used providers, MEGA and Nextcloud [3, 4, 8, 10, 40, 47]. This raises the question: are there providers in the broader ecosystem that have better security guarantees than MEGA and Nextcloud? After analysing five major providers, we answer this question in the negative for four of them by unveiling attacks against their systems. In particular, we highlight flawed anti-patterns that are common across the providers we analysed and that introduce vulnerabilities in their protocols.

1.1 Our Contributions

We contribute a detailed analysis of several cloud storage systems that claim to provide end-to-end encryption, focussing on storage systems that either have relevant enterprise-level customers or are estimated to have a large user base. Namely, we analyse Sync, pCloud, Icedrive, Seafile and Tresorit, collectively serving more than 22 million customers, storing hundreds of petabytes of data and managing files in the order of billions [34, 44, 48]. These providers frequently emerge at the top of internet searches for queries such as "best end-to-end encrypted cloud storage" and "most secure cloud storage". Given the influence such search results likely have

on the average non-technical person, we consider our selection of providers to be representative of the options most commonly chosen by users. While, clearly, there are always more providers to be analysed, we leave them to future work. Indeed, our examples, combined with the existing analyses of MEGA and Nextcloud, already showcase critical vulnerabilities in a large majority of the field and thus provide an adequate basis to raise concerns about the current state of E2EE cloud storage.

We analyse the end-to-end encryption of these providers in the natural setting of a compromised server. This is a fair expectation for E2EE cloud storage: security should be preserved even if the attacker has full access to the server and can directly interact with the user. This threat model also aligns with the advertisement of various cloud providers, as many of them claim to be unable to access user data thanks to what they dub "zero-knowledge encryption".

Not all of our attacks are sophisticated in nature, which means that they are within reach of attackers who are not necessarily skilled in cryptography. Indeed, our attacks are highly practical and can be carried out without significant resources. Additionally, while some of these attacks are not novel from a cryptographic perspective, they emphasise that E2EE cloud storage as deployed in practice *fails at a trivial level* and often does not require more profound cryptanalysis to break.

Our attacks are not limited to violating confidentiality but rather focus on many aspects that are relevant to cloud storage systems. In particular, in addition to the confidentiality of file contents, we target metadata (with a focus on file names and location) as well as the integrity and authenticity of files. In total, we present ten classes of attacks, split into four categories.

We find four classes of attacks that violate confidentiality:

(1) *Lack of authentication of user key material.* In Sync and pCloud the adversary can exploit the absent authentication of public-key encrypted ciphertexts to inject adversary-controlled keys, which the client will use to encrypt their data. Furthermore, in pCloud, the adversary can abuse the unauthenticated symmetric encryption scheme to overwrite parts of the private key. This allows overwriting the user's key with an attacker-controlled key, which will be used to encrypt all files uploaded from that point onwards.
(2) *Unauthenticated public keys.* Sync and Tresorit use per-user public keys to share files with other users. These public keys are provided by the (possibly malicious) server and, thus, need to be authenticated. We have observed that neither Sync nor Tresorit provides such an authentication mechanism.
(3) *Protocol downgrade.* In Seafile, the adversary can downgrade the security of the encryption protocol, which allows it to attempt brute-force of user passwords.
(4) *Link-sharing pitfalls.* In Sync, the client shares a file by sending a link which encodes the password needed to decrypt. Whenever the link is clicked, it sends the password to the server, trivially breaking confidentiality.

Two more attack classes target file data:

(5) *Unauthenticated encryption modes.* The usage by Icedrive and Seafile of unauthenticated cipher modes such as CBC allows an attacker to tamper with the content of files in a semi-controlled manner.
(6) *Unauthenticated chunking.* In Seafile and pCloud, files are chunked prior to encryption, but the list of chunks is not properly authenticated, allowing an adversary to swap chunks around and remove chunks from files.

For metadata, we provide two classes of attacks:

(7) *Tampering with file names and location.* None of the providers, except for Tresorit, have mechanisms to cryptographically bind file locations and file names to the contents, allowing an adversary to move files to different locations in the storage and swap their names. In some cases, the adversary can also tamper with the names of the files.
(8) *Tampering with file metadata.* For all providers, metadata (e.g. file size, file type, and modification date) is not authenticated, which allows an adversary to change it arbitrarily.

Finally, we present two classes of attacks that can allow an attacker to inject files into the user's storage, making it appear as if the user had uploaded them:

(9) *Injection of folders.* In Sync, a malicious server can make a folder appear as if the user uploaded it by exploiting the peculiarities of the sharing mechanism and by building on top of our metadata-editing attack, which makes a shared folder appear as if it was a standard folder.
(10) *Injection of files.* For pCloud, we use the lack of authentication of RSA ciphertexts to inject rogue file keys, along with rogue file content in the user's storage.

Additionally, we point out how some providers frequently leak information about files, such as their length or similarity (e.g. through the usage of deterministic encryption). Many providers also leak the metadata of files, as well as the directory structure, and where each file is located. We do not consider these to be attacks *per se*, as a malicious server need not take any action to achieve this leakage. At the same time, we consider them to be severe shortcomings that can impact security for users.

In summary, our attacks are representative of a wide gap in the security guarantees of these providers and showcase fundamental design flaws in real-world E2EE cloud storage systems. Our analysis complements the existing work on MEGA and Nextcloud, as it shows that cryptographic shortcomings are not limited to a few providers but are, instead, widespread in the broader ecosystem.

We conclude with a discussion on the security of E2EE cloud storage. First, we collect the anti-patterns that emerged from our investigation. Many of these failure modes are well-known in the cryptographic community, and they can be largely avoided by using standardised primitives. Despite this, our investigation highlights how they still appear in real-world products. For each anti-pattern, we also contribute with mitigations that would improve security in the short term. Second, we discuss the steps needed to improve the ecosystem, focussing on the analysis and standardisation of a secure E2EE cloud storage protocol.

1.2 Related Work

Analyses of Cloud Storage Systems. MEGA has been the subject of scrutiny of previous works [4, 8, 47], which found severe vulnerabilities in MEGA's bespoke cryptographic design. Among others,

a malicious server could inject files in the personal storage of users and recover their private keys.

In a blog post, Böek [10] presents an attack on the server-side encryption of OwnCloud, exploiting their usage of AES-CFB, an unauthenticated block cipher mode, in order to backdoor a Windows executable. Niehage, in [40], finds vulnerabilities in Nextcloud's server-side encryption, showing that an attacker could swap chunks of files, inject new files, and tamper with file data. Albrecht et al. [3] study the client-side encryption of NextCloud, where they show that the lack of authentication of RSA ciphertexts allowed the server to decrypt user data and inject files. Many variants of these attacks re-emerge in our analysis.

Key Overwriting Attacks. In E2EE cloud storage, it is common to store encrypted cryptographic material on a (possibly untrusted) server. This setting has been studied for OpenPGP by Klima and Rosa in [29] and later by Bruseghini et al. [11] and led to the discovery of a new class of attacks called Key Overwriting (KO) attacks. We provide a novel variant of a KO attack where the adversary, rather than attempting to recover the private key, forces the user to use an attacker-controlled keypair for encryption and decryption.

Creating Secure E2EE Cloud Storage. On the constructive side, a survey by Virvilis et al. [50] lists various security properties that are desirable from cloud storage providers, including confidentiality and integrity. In their work, they highlight the importance of a holistic security approach when developing secure cloud storage. Backendal et al. [7] present the main challenges for developers of E2EE cloud storage systems, including key management and file sharing, and call for the standardisation of a secure E2EE cloud storage protocol, which they argue will require a joint effort between cryptographers, vendors and implementers. Recently, Backendal et al. [6] proposed a formal model to capture the security of E2EE cloud storage systems, as well as a protocol that provably achieves the desired security properties. Our work is complementary to such efforts, as it challenges the notion that cloud storage is a "solved problem". It is, in fact, a field where many providers fail, often trivially, to provide the security guarantees they claim to offer.

1.3 Ethical Considerations

We have notified Sync, pCloud, Seafile, and Icedrive of our findings on 23.04.24, proposing a coordinated disclosure of the vulnerabilities and suggesting the standard 90 day disclosure window. The Icedrive team acknowledged our email on the same day and, after a brief exchange, opted not to address the issues we raised. The Seafile team acknowledged the email on the 24.04.24, and replied on the 29.04.24, informing us that they will patch the protocol downgrade issue by forcing version 2 to be used, and stating that storage integrity is not in the scope of their design. As of 04.09.24, Sync and pCloud have yet to respond to multiple attempts to contact them through different channels. We did not contact Tresorit as we had no major issue to report.

For Seafile, we analysed the open-source code of the desktop application, as the web application sends the user password to the server and thus does not provide end-to-end encryption. For all other providers, we analysed the JavaScript code that was executed in the browser and that was directly available to us using the browser development tools. Sync additionally provides the non-minified source code of their web application in the browser, which we used for our investigation. We did not disassemble or decompile any application and we did not attempt to reverse engineer any server functionality. For testing, we have exclusively used accounts under our control and we have avoided any action that would have put excessive load on the provider's infrastructure or that would have affected other users. To the best of our knowledge, we did not violate the terms of service of any of the providers.

We are not affiliated with any of the providers we analyse, nor any of their competitors.

1.4 Paper Structure

We describe the protocol and key hierarchy of all five providers in Section 2. We then present our attacks against these providers in Section 3. In Section 4, we abstract the common problems with current implementations of end-to-end encrypted cloud storage, drawing wider lessons from our analysis. Finally, we give our conclusions in Section 5.

2 DESCRIPTION OF THE PROVIDERS

2.1 Cryptographic Primitives and Notation

In the remainder of the paper, we use a number of symmetric keys (for encryption or MAC), which we denote by K and that we label using a subscript (e.g. K_{master}, K_{file}). Asymmetric keypairs (for encryption) are denoted by (sk, pk), indicating the private key and the public key, respectively.

Symmetric encryption primitives are described by combining a block cipher with a block cipher mode of operation and the relevant operation. For example, AES encryption in Cipher Block Chaining (CBC) mode would be represented by the function AES.CBC-Enc. The order of parameters is key, data to encrypt/decrypt, IV (if applicable) (e.g. $\text{AES.CBC-Enc}(K_{\text{master}}, m, \text{IV})$). For HMAC, we always specify the underlying hash function. For instance, $\text{HMAC-SHA512}(K, m)$ is the HMAC algorithm with SHA512 as the hash function, using key K and applied on message m.

With $\oplus$, we refer to bitwise xor. String concatenation is written as $\|$, while $|x|$ refers to the length of x in bytes. We use Python-style slicing notation for strings and arrays, with indices starting from 0: $s[a : b]$ indicates a string composed of all elements from the a-th element of s to the $(b-1)$-th element (inclusive). If a or b are omitted, they assume the values 0 and $|s|$, respectively.

2.2 Description of the Protocols

The security of all the cloud storage systems we analyse pivots on a user-chosen password $\mathcal{P}$ (often distinct from the one used to authenticate to the server). This is a natural choice, as the user should be able to access their account from multiple devices and having to transfer high-entropy cryptographic material across devices would negatively impact user experience. However, a password is unsuitable to be used directly for cryptographic operations, so a Key Derivation Function (KDF) like PBKDF2 or scrypt is used to obtain the key material needed for the following steps of the protocols.

Because of the reliance on password-derived key material, the security of the systems we analyse is highly dependent on the strength of the user's password. A malicious provider can always attempt

an offline brute-force attack or a dictionary attack to recover a password, which is a fundamental limitation of password-based encryption. Good password policies and the usage of a memory-hard password hashing function help to mitigate this risk.

When describing key hierarchies, we distinguish between cryptographic keys which are used to encrypt other keys (Key Encrypting Keys, or KEK), keys which are used to encrypt data (Data Encrypting Keys, or DEK), and keys which are used to encrypt metadata (Metadata Encrypting Keys, or MEK). In general, we simplify the protocol descriptions by glossing over details that are irrelevant for our analysis or attacks (e.g. encoding format).

We represent the key hierarchies for all providers in Fig. 1. Additional information about the folder structure of each provider can be found in Appendix A.

2.2.1 Sync. Sync is a Canadian company founded in 2011, which offers its cloud storage services to over 2 million users worldwide, including entities such as the government of Canada, the university of Toronto, and the Canadian red cross, and stores more than 130 petabytes of data [48]. The company offers a web application, as well as desktop and mobile clients. In our analysis, we focus only on the web application.

Cryptographic Primitives. For symmetric encryption, Sync uses AES-GCM with random IVs. Asymmetric encryption uses RSA with PKCS1v1.5 padding. Sync makes use of PBKDF2-SHA256 with a random 12 byte salt value as KDF.

Key Hierarchy. During the registration process, the Sync client derives two 32 byte KEKs K_{master} and K'_{master} from $\mathcal{P}$ using the KDF with different salt values. The client then samples a new RSA keypair (sk, pk) and uses K_{master} to encrypt it. The client also samples a 32 byte symmetric MEK K_{meta} (encrypted under K_{master}) and a 32 byte symmetric KEK K_{share}, called the share key (encrypted under pk). All the ciphertexts are then offloaded to the server. Whenever the client logs in, the key material is fetched, decrypted, and stored in the browser.

When uploading a file, its contents are symmetrically encrypted with a freshly generated DEK K_{file} and the file name is encrypted under the user's MEK. Then, K_{file} is encrypted using the share key K_{share} and, finally, the encrypted DEK, the encrypted file data, and the encrypted file name are offloaded to the server. Folder names are also encrypted using the MEK and sent to the server.

Sharing. Sync allows files to be shared by using either *link sharing* or by *permanent sharing* of a folder. Link sharing allows anyone in possession of a special link to access a file. To achieve this, the user randomly samples a string of 32 alphanumerical characters, called the link password $\mathcal{P}_{\text{link}}$. A link share key K_{link} is then derived from $\mathcal{P}_{\text{link}}$ using the KDF and used to encrypt the file name and K_{file}. These two ciphertexts are then uploaded to the server. The password $\mathcal{P}_{\text{link}}$ is included in the share link as part of the URL path, which allows other users to derive K_{link}, and thus decrypt the file.

Sync also allows for folders to be permanently shared. When a user $\mathcal{A}$ wants to create a folder and share it with user $\mathcal{B}$, $\mathcal{A}$ samples a new KEK $\tilde{K}_{\text{share}}$ and uses it to symmetrically encrypt the DEKs of all files in the folder. These encrypted DEKs are uploaded to the server. The share key $\tilde{K}_{\text{share}}$ is encrypted against $\mathcal{B}$'s public key, which $\mathcal{A}$ obtains by querying the server, and then forwarded by the server via email to $\mathcal{B}$. We note that $\mathcal{B}$'s key is not authenticated, an issue which we elaborate on in Section 3.1.2.

2.2.2 pCloud. The Swiss-based pCloud is the largest provider that we consider, with over 19 million users [43]. Upon signing up, users are required to set up an additional password specifically for accessing their encrypted storage, distinct from their regular authentication password. We ignore the authentication password, as it is not involved in cryptographic computations, and call the additional password $\mathcal{P}$.

Cryptographic Primitives. For symmetric encryption, pCloud uses different primitives depending on the type of data encrypted. When using a symmetric KEK to encrypt a private key, pCloud uses a custom variant of counter mode, depicted in Fig. 2. When using a symmetric DEK to encrypt file data, pCloud uses a bespoke block cipher mode which uses a synthetized IV. When using a symmetric MEK to encrypt file names, a different variation of CBC mode is used. We describe the pseudocode for the last two procedures in Algorithm 1.

For asymmetric encryption, pCloud uses RSA with OAEP padding, with SHA1 as the hash function. Key derivation is done using PBKDF2-SHA512 with a 64 byte random salt.

Key Hierarchy. Upon registration of a new user, the password $\mathcal{P}$ is used to derive a 32 byte symmetric key K_{master} and a 16 byte $\text{IV}_{\text{master}}$. Furthermore, an RSA keypair (sk, pk) is generated and the private key is encrypted with K_{master} and $\text{IV}_{\text{master}}$ using the custom variant of counter mode in Fig. 2.

Whenever the client wants to upload a file, they retrieve the encrypted private key sk, which they decrypt using their password, and the (unauthenticated) public key pk from the server. Since the private key and the public key have been retrieved separately and the public key is not authenticated, the client has to check for their consistency. pCloud does so by making the client sample a random hexadecimal string of 32 characters, which it then encrypts with pk and decrypts with sk, verifying in the end that the original string is returned. If this check is successful, the client generates a 32 byte symmetric encryption key $K^{\text{file}}_{\text{enc}}$ and a 128 byte HMAC key $K^{\text{file}}_{\text{HMAC}}$. The file is then split into sectors, each of 4096 bytes except possibly for the last one. Each sector is encrypted under $K^{\text{file}}_{\text{enc}}$ and $K^{\text{file}}_{\text{HMAC}}$, using the procedure pCloudSectorEncrypt in Algorithm 1, and yielding a ciphertext-tag pair (c, τ). To provide integrity for the file, all tags are included as leafs of a 128-ary Merkle tree, where each internal node is created by computing the HMAC-SHA512 of (up to) 128 tags from the layer below. The process stops when a layer consisting of only one node is created. Each layer of the Merkle tree is included alongside the file and checked for integrity when the file is downloaded.

When creating a folder, pCloud generates two keys, $K^{\text{folder}}_{\text{enc}}$ and $K^{\text{folder}}_{\text{HMAC}}$, which will be used to encrypt the names of the contained files and subfolders, using pCloudNameEncrypt in Algorithm 1.

Sharing. pCloud does not support sharing encrypted files.

2.2.3 Icedrive. Icedrive is the youngest provider we analyse, as its cloud storage product was released in 2019. It has been estimated to have as many as 150'000 customers [23]. They provide a web

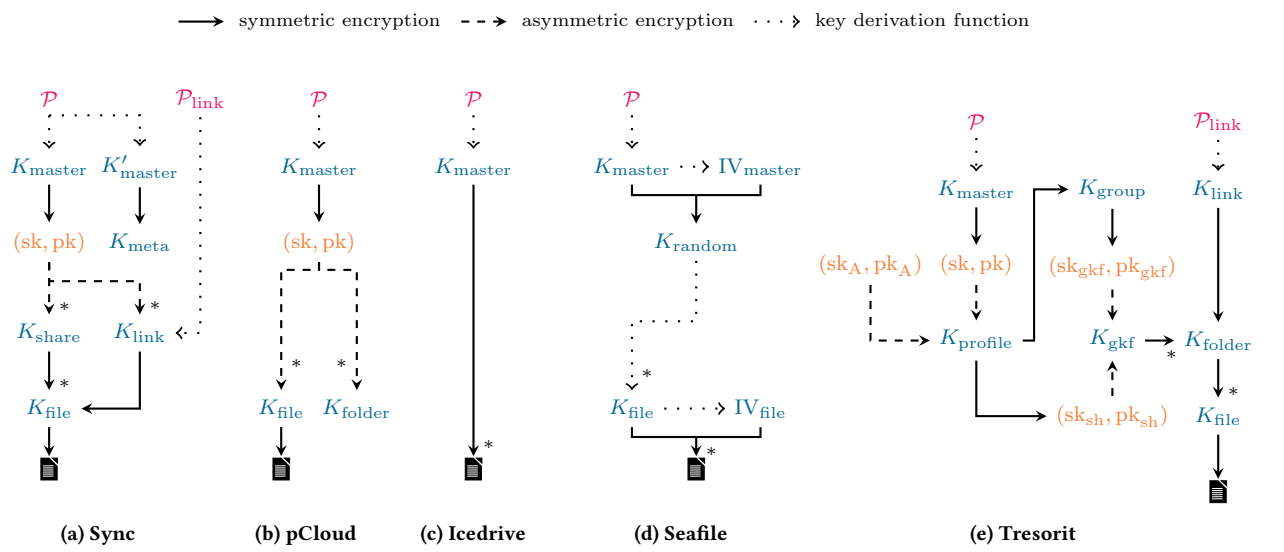

Figure 1: Key hierarchies of all providers investigated. Arrows marked with $*$ indicate a one-to-many relationship. All the key material that is not derived from something else is implicitly generated using an appropriate key generation function.

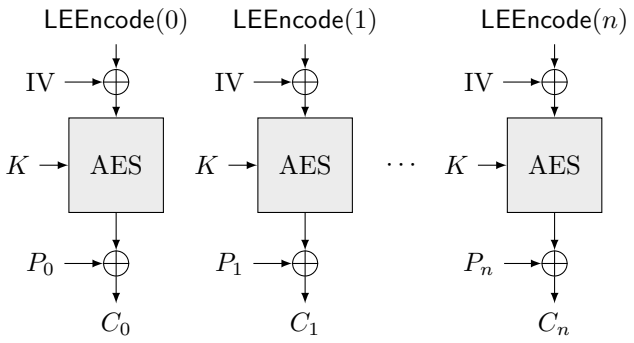

Figure 2: The CTR-based block cipher mode used by pCloud using key K and initialization vector IV, encrypting blocks of plaintext P_i (16 bytes in size, except for possibly P_n) and obtaining the ciphertext blocks C_i. The function LEEncode computes the 16 byte little-endian encoding of the input.

Algorithm 1 pCloud's encryption procedure for files

procedure pCloudNameEncrypt(K_{enc}, K_{HMAC}, m)
  $m \leftarrow m \,\|\, 0^{8\cdot(16-(|m| \mod 16))}$ *// zero-padding*
  if $|m| = 16$ **then**
    return AES.ECB-Enc($K_{enc}, m \oplus K_{HMAC}[:16]$)
  IV $\leftarrow$ HMAC-SHA512($K_{HMAC}, m[16:])[:16]$
  return AES.CBC-Enc(K_{enc}, m, IV)

procedure pCloudSectorEncrypt($K_{enc}, K_{HMAC}, m, i_{sector}$)
  $\rho \leftarrow \$\{0,1\}^{128}$
  IV $\leftarrow$ HMAC-SHA512($K_{HMAC}, m \,\|\, i_{sector} \,\|\, \rho)[:16]$
  $c \leftarrow$ AES.CBC-Enc(K_{enc}, m, IV)
  $\tau \leftarrow$ AES.ECB-Enc($K_{enc}, \rho[:8] \,\|\, \text{IV} \,\|\, \rho[8:]$)
  return (c, τ)

application, which we focus on in our analysis, as well as clients for desktop and mobile.

Cryptographic Primitives. Icedrive has an unusual choice of block cipher, as they employ TwoFish and motivate this choice by claiming that TwoFish is "widely accepted by cryptographers as a more secure solution than AES/Rijndael" [32]. The block cipher is used to encrypt files in a custom block cipher mode, which we depict in Fig. 3. When encrypting file and folder names, a standard CBC construction is used, albeit with a fixed IV. For key derivation, Icedrive uses PBKDF2-SHA256.

Key Hierarchy. In Icedrive, the password $\mathcal{P}$ is used in conjunction with PBKDF2-SHA256 and a random salt, stored on the server, to obtain a 32 byte symmetric key K_{master}, used as both a DEK and a MEK. Prior to encryption, files are split into chunks of 2 megabytes and each chunk is encrypted with the same key and IV using the procedure described in Fig. 3, with TwoFish as the underlying block cipher and IV′ = "1234567887654321". Note that files are padded with zero bytes, with the length of the padding in bytes λ encoded in the header, which makes the padding process reversible. The unpadding procedure, however, does *not* check whether the bytes of padding are zero, so the last λ bytes are removed, regardless of their value. The use of the same IV for all chunks, means that chunks which have (at least) the first 16 bytes in common will also share the first blocks of ciphertext, revealing their similarity to the adversary. We also note that, even though file encryption uses a randomized IV, the process samples the IV from a restricted set of characters (letters and digits), rather than from $\{0,1\}^{128}$. A formal analysis similar to the one in [9] shows that this peculiar

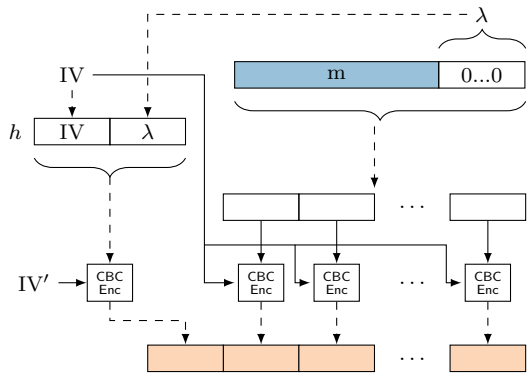

Figure 3: The encryption procedure for files in Icedrive. The message m (in blue) is zero-padded and split into chunks of 2^{22} bytes, except for the first ($2^{22} - 32$ bytes), and possibly the last one. Chunks are encrypted with K_{master} (omitted) and a random IV. The IV and the padding length λ are encrypted using K_{master} and a *fixed* IV′ and prepended to the encrypted chunks to obtain the ciphertext (in orange). Note that the header h is deterministically encrypted.

choice weakens the IND-CPA security bound of AES-CBC as used in Icedrive, as it increases the probability of an IV collision, which can be detected due to the deterministic encryption of the header.

File and directory names are encrypted using TwoFish.CBC-Enc under K_{master} and the fixed IV′. The considerations above regarding the fixed IV also apply here.

Sharing. Icedrive does not support sharing encrypted files.

2.2.4 Seafile. Seafile differs from the other cloud storage providers, as its code is open-source, for both client and server. Much like Nextcloud, they do not host server instances themselves, but rather allow users and companies to host their own servers. They boast 11k stars on GitHub [22] and serve more than 1 million users, including organisations such as Kaspersky, the Humboldt University in Berlin, the University of Strasbourg and the University of Turku. Seafile offers a web interface, a desktop application and a mobile application. The web application is not truly end-to-end encrypted, as it sends the password to the server for decryption. This shortcoming is advertised in the Seafile admin manual [21] and is the reason why we focus only on the desktop client.

Cryptographic Primitives. The cryptographic primitives used by the Seafile client depend on the version of the encryption protocol. For backwards compatibility, the client supports all versions. We provide all the primitives used in each version in Table 1. The BytesToKey algorithm, as specified by the OpenSSL documentation [5], consists of repeated application of a chosen hash function H and can be recursively described as $D_i = H^k(D_{i-1} \parallel \text{data} \parallel \text{salt})$, where D_0 is the empty string, k is the number of iterations, and H^k indicates $H(H(...H(\cdot)))$ nested k times. Values $D_1, D_2, \ldots$ are computed and concatenated until enough cryptographic material has been obtained.

To choose the version, the client queries the server and chooses the implementation contained in the server's response. Version 2 is the default and is used by Seafile's official demo server and, thus, we describe the behaviour of the client only for that version.

Table 1: Summary of all the primitives used by Seafile on all supported versions.

Version	Encryption	Key Derivation Function		
		Algorithm	Iterations	Salt
0	AES-256-CBC†	BytesToKey-SHA1	3	None
1	AES-128-CBC†	BytesToKey-SHA1	2^{19}	Fixed‡
2	AES-256-CBC†	PBKDF2-SHA256	1000	Fixed‡
3	AES-128-ECB	PBKDF2-SHA256	1000	Random

†The IV is fixed and used for multiple encryptions.
‡The salt is hard-coded in the client: `0xda9045c306c7cc26`.

Key Hierarchy. The Seafile storage model consists of *repositories*, each associated with its independent cryptographic material. In particular, rather than having a per-user password, users provide one password per repository.

During the creation of the repository, the password $\mathcal{P}$ is passed through the KDF to obtain a 32 byte symmetric KEK K_{master}. Then, K_{master} is passed through the KDF again to obtain a 16 byte value $\text{IV}_{\text{master}}$. A 32 byte symmetric KEK K_{random} is generated and its encryption under K_{master} and $\text{IV}_{\text{master}}$ is sent to the server. Another step of key derivation with K_{random} as input yields the DEK K_{file}, which is used to derive IV_{file}.

Prior to encryption, files are split into chunks in order to support deduplication. The chunking procedure uses content-defined chunking [17]. Each chunk is encrypted using K_{file} and IV_{file}. Similarly to Icedrive, the reuse of a fixed IV leads to leakage of similarities between chunk plaintexts: if two chunks begin with the same 16 bytes, their ciphertexts will also share the first blocks.

Seafile provides a way for the user to check whether they have input the correct password when accessing the repository. The client uses the KDF, with the ID of the repository concatenated with the user password as input, yielding a 32 byte value called the "magic" string, which is sent to the server. Whenever the user provides a password, the magic string is recomputed and checked against the server-provided value. If they match, the client proceeds to decrypt the files in the repository.

Sharing. An entire repository can be shared by sending the password to another user via an out-of-band channel and then giving the other user a link to access the repository.

2.2.5 Tresorit. Tresorit was founded in 2011 as a provider mainly geared towards businesses. Currently, the Swiss Post holds a majority stake in the company. Tresorit has released a whitepaper containing the technical details of their protocol [2], which we supplement by inspecting the client source code. The cryptographic design of Tresorit is remarkably more complex than the other providers, partially due to the advanced features that it provides, such as password recovery and admin access for user accounts.

Cryptographic Primitives. Tresorit uses AES-GCM to encrypt key material with random IVs. File encryption is done using AES in OpenPGP-style CFB mode [19] in an Encrypt-then-MAC composition with HMAC-SHA512. For asymmetric encryption, Tresorit

makes use of RSA-OAEP with 4096 bit moduli. For key derivation, Tresorit uses scrypt, as well as PBKDF2, with a 32 byte salt.

Key Hierarchy. During registration, the user derives a symmetric KEK K_{master} from $\mathcal{P}$. The master key K_{master} is used to encrypt a freshly generated RSA keypair (sk, pk). Then, pk is used to encrypt a symmetric key K_{profile} that, in turn, encrypts the so-called user profile which acts as a container for the keys associated to the user.

Tresorit organizes files in *Tresors*, which are top-level folders, each associated with a group key file (GKF). A GKF contains the keys for the root directories inside the Tresor. The GKF is encrypted under a KEK K_{gkf}. Mirroring the encryption of the user profile, K_{gkf} is encrypted using an asymmetric key $(\text{sk}_{\text{gkf}}, \text{pk}_{\text{gkf}})$, which is in turn encrypted under a KEK K_{group}, which is stored in the user profile and thus will be encrypted under K_{profile}. The GKF contains a DEK K_{root} for each top-level folder in the Tresor, which is used to encrypt folder-related data. Folders are implemented as files containing the names, URL and key material of all files.

Tresorit uses a KDF in conjunction with both K_{folder} and K_{file} to derive an AES key and an HMAC key from each key. The derived keys are subsequently used for encryption and authentication of folder or file data, respectively. When a folder is changed (e.g. a file is added), two new salt values are randomly sampled and fresh AES and HMAC keys are generated from K_{folder} to reencrypt the folder.

The user profile contains a "key history", which is used to guarantee its authenticity by binding it to the user password. This prevents attacks in which the server substitutes the user key material with keys under its control (cfr. Section 3.1.1). We omit details for this mechanism, as they are not relevant for our analysis.

All asymmetric key material in Tresorit is authenticated using public-key certificates signed by Tresorit's own certification authority. The certificates are validated whenever the client requires public key material from the server.

Tresorit allows company admins to inspect, manage and delete the accounts of users, by making them store on the server an additional encryption of K_{profile} under the admin's public key pk_{A}. Before this, the user must explicitly acknowledge a pop-up message that contains the fingerprint of pk_{A}, for out-of-band verification.

Sharing. The Tresorit protocol also provides a file sharing functionality. Similar to Sync, we differentiate between link sharing and permanent sharing. For link sharing, the user generates a 16 byte client secret, which is then used to derive a key K_{link}, as well as an identifier, using the KDF. The identifier is used to fetch and decrypt the file associated with the parent directory of the target file, from which the client obtains the file name and the corresponding K_{file}. The client is then able to decrypt the file content.

Tresors can be shared permanently. To do so, the client retrieves the public key pk_{sh} of the invitee from the server. This public key is then used to encrypt K_{gkf} of the Tresor to be shared. The encrypted key material is then sent to the server, who relays it to the invitee. Each member of a share stores the secret sk_{sh} in their own user profile, allowing them to access the Tresor at will. Note that the public key pk_{sh} is authenticated using a certificate signed by Tresorit's CA.

3 ATTACKS

We provide several attacks against all providers, many of which affect multiple protocols in the same way. In the malicious server setting, the adversary's objectives include violating the confidentiality of files. However, we believe that the notion of security should extend beyond just keeping the contents of files confidential. Indeed, it is commonly understood that metadata such as the file name can disclose much about the file itself. Additionally, the server may wish to tamper with the metadata of files or their location, which also encode semantic information about the files. Lastly, an adversarial server may wish to inject files in order to confuse the user. All these attacks diminish the control that the user has of their own data, and thus have to be protected against when developing a truly secure E2EE cloud storage system.

We categorize the attacks in four classes: (1) Attacks that allow the server to learn information about the file contents or names and the names of directories (Section 3.1), (2) attacks that affect the integrity of the files, exploiting non-authenticated cipher modes or unauthenticated chunking of files (Section 3.2), (3) attacks that tamper with information of files which is unrelated to their content, including location of files in the storage, file names, and metadata (Section 3.3), and (4) attacks that allow the server to insert targeted files which decrypt correctly and look as if they were uploaded by the user itself (Section 3.4).

We provide an overview of providers and attacks in Table 2.

As of April 2024, all our attacks have been validated and work against the latest version of each client.

One interesting observation is that no provider reports cryptographic errors to the server, silently crashing on the client side instead and without automatically retrying operations. This means that cryptographic oracles are created only when the user directly interacts with the system. For example, a failed cryptographic operation might lead to the web client not showing files which it could not decrypt, in which case the user might try to manually refresh the page. In this case the server would see two requests for the same page in a short timespan, from which the adversary might infer that a cryptographic operation failed. This slow and burdensome flow hinders attacks that require many queries. For instance, we do not consider padding oracle attacks on CBC mode to be viable, in the providers we analyse. At the same time, the theoretical possibility of such attacks showcases severe weaknesses in the protocol, which beckons for countermeasures to be taken.

For our analysis, we ignore denial-of-service attacks, since a provider can always refuse to serve files and it is a challenging issue to mitigate by cryptographic means.

We also ignore attacks where the provider supplies malicious JavaScript to the user, as no widespread mechanisms exist for preventing such attacks at the moment. Indeed, a malicious server can trivially serve a web page that has been modified to leak the user password, which is an inherent obstacle to secure end-to-end encryption on the web. While general proposals exist to mitigate this [25, 28], they lack broad adoption and browser support. As a provider-specific countermeasure, MEGA has developed a browser extension that loads code from the extension rather than from the server. Still, very few users can adequately audit code updates, and

the server can target select users with malicious updates. Additionally, even diligent code audits may miss ways for the server to dynamically injecting malicious code. In fact, web applications often assume that data coming from the browser need not be sanitized, an assumption which does not hold in the E2EE setting. Interestingly, this means that the server can try to inject cross-site scripting (XSS) payloads on the client in order to exfiltrate secret keys, an uncommon setting for XSS, as the server is usually trusted. Indeed, we have discovered that Icedrive has a few instances in the code where the value of `innerHTML` is set to server-provided data without sanitization, allowing for XSS.

Nonetheless, our attacks target the cryptographic protocols directly and, as such, showcase weaknesses on a more fundamental level. Most of the attacks do not depend on any implementation details, but rather on the protocol design itself.

3.1 Confidentiality Violation

We provide practical attacks against Sync, pCloud and Seafile, which target the cryptographic material and, as a consequence, violate the confidentiality of contents or metadata of user files (Section 3.1.1). In particular, for Sync, a malicious server is able to force the client to encrypt files using an attacker-controlled key, which then enables the attacker to decrypt them. For pCloud, our attack leverages the lack of authentication of the encrypted RSA private key to force the client to use an attacker-controlled RSA keypair. In addition, we exploit permanent sharing in Sync and Tresorit, where the lack of a public key infrastructure or out-of-band verification allows an adversary to violate the confidentiality of shared folders (Section 3.1.2). For Seafile, we provide a downgrade attack that weakens the KDF used by the client, giving the server the opportunity to brute-force the user password (Section 3.1.3). An additional attack on Sync uses the fact that, when sharing files, the sharing password is leaked to the server, which trivially violates confidentiality (Section 3.1.4).

3.1.1 Sync, pCloud: Unauthenticated Key Material. Many providers use RSA to encrypt cryptographic keys, which are then (indirectly) used to encrypt files. When using a solid primitive, like RSA-OAEP, the attacker is unable to directly decrypt the keys. However, RSA-OAEP provides confidentiality *but not authentication*. Indeed, since the server has access to the public key, it can also create valid ciphertexts of arbitrary messages. In particular, the server can encrypt arbitrary cryptographic material and substitute it for the user's. When the user retrieves the encrypted cryptographic material, they have no means to verify whether the ciphertext is authentic and will use the server-chosen cryptographic material to encrypt their data. The server is then able to decrypt all data that has been uploaded after the substitution has taken place. We dub this a *key replacement attack*.

We put this attack into concrete terms by explaining how it affects the design of Sync and pCloud. Recall that, in Sync, the public RSA key pk is used to encrypt the user's share key K_{share}, which is then used as a KEK for all the file keys. Substituting the share key with a server-controlled share key (for example at registration time) is sufficient for the server to decrypt all files uploaded from that point on. Similarly, for pCloud, the server can replace folder keys, since they are encrypted with the public RSA key of the user and are not authenticated. The adversary can target a specific folder, generating new keys $K_{\text{enc}}^{\text{folder}}$ and $K_{\text{HMAC}}^{\text{folder}}$, and encrypting them with the public key of the user. All files uploaded in that folder from that point on will have their names leaked to the server.

An additional attack vector is given by the lack of integrity of the asymmetric keypairs, for example in the case of pCloud. Our attack here is a novel variation of a key-overwriting attack that leads the client to use an attacker-chosen public key for encrypting file keys. For the attack, we exploit the fact that the user's public key is unauthenticated and that the private key is encrypted using an unauthenticated stream cipher mode.

The objective of the attacker is to change the value of the user's public key to a value of which it knows the corresponding private key. Overwriting only the public key is not possible, due to the consistency check implemented by pCloud between the public and private key. However, the unauthenticated counter-like cipher mode used by pCloud to encrypt the private key allows an attacker to arbitrarily flip bits in the plaintext by flipping the corresponding bits in the ciphertext. The idea of the attack is that the adversary can first set the public key to an attacker-controlled value of which they know the corresponding private key. Afterwards, they can malleate the encrypted private key to make it consistent with the public key, in order to pass the check. If the check passes, all file keys from that point on will be encrypted with an attacker-chosen public key, violating confidentiality.

Since only the known parts of the private key can be malleated in a targeted way, there is apparently no way of setting values for the private attributes. More precisely, private RSA keys are encoded into DER format [27], depicted in Fig. 4. In this format, the adversary knows all headers (as they only encode the data type and length) and the values of N and e, for a total of 532 bytes. The main insight of the attack is that this space is sufficient for embedding the DER encoding of a private key *with a small modulus*. By setting the first header to encode the length of the new key, rather than the old one, the attacker can make the DER decoder ignore the trailing data and return only the embedded private key.

There are two obstacles to this attack, in practice. The first one is that many implementations of the Web Cryptography API [26], which pCloud uses for their web application, implement checks on the well-formedness of RSA keys. In particular, they check that $p \cdot q = n$, $3 \leq e < 2^{64}$, $e \cdot d = 1 \mod \text{lcm}(p-1, q-1)$, and that the values of $d \mod p-1$, $d \mod q-1$, and $q^{-1} \mod p$ are correct. The second obstacle is that the RSA key is used to encrypt file and folder keys, whose encoding consists of 8 bytes of header, 32 bytes of encryption key and 128 bytes of HMAC key. The unconventional choice of key length for the MAC key means that the padding procedure will require a modulus of at least 210 bytes in size.[1] Due to the constrained space, we are unable to create a key that the web client will accept and that allows encryption of messages of such size. However, these checks are *implementation-dependent*. For example, the Web Cryptography API does not mandate them and they are, instead, inherited from the underlying OpenSSL library. Other implementations of RSA-OAEP might not perform such checks. Indeed, the library used

[1] 210 bytes = 42 (for the OAEP padding) +8 (header) +32 (K_{enc}) +128 (K_{HMAC}) bytes.

Table 2: Summary of the providers analysed, with the attacks and leakages that affect them.

● Attack works ◐ Attack works under specific conditions ○ Attack does not work – The attack is not applicable

	Unauthenticated Key Material	Unauthenticated Public Keys	Protocol Downgrade	Link-sharing Leakage	Unauthenticated Encryption	Unauthenticated Chunking	Tampering with Files and File Names	Tampering with Metadata	Folder Injection	File Injection	Leaks Plaintext Information	Leaks Metadata	Leaks Directory Structure
Sync	●	●	○	●	○	○	●	●	●	◐ *	○	●	●
pCloud	● †	–	○	–	○	●	●	●	●	●	○	●	●
Icedrive	○	–	○	–	●	●	●	●	○	◐ **	○	●	●
Seafile	○	–	●	–	●	●	●	●	●	◐ **	●	●	●
Tresorit	●	●	○	○	○	○	○	●	○	○	○	●	●

†Works in the CLI client. Most browsers implement adequate checks for public keys which prevents the attack in that setting.
*Only as a consequence of folder injection.
**The adversary can only create a new file by composing chunks of other files, hence the attack is not targeted.

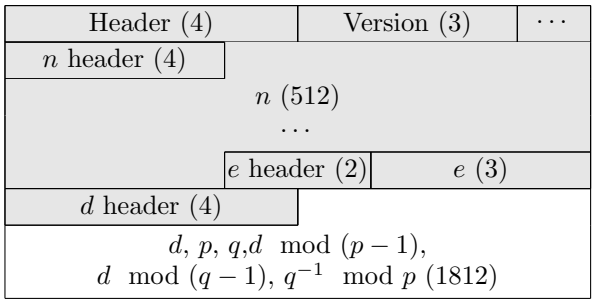

Figure 4: The structure of a DER encoded RSA private key, with the length of each component in bytes. All publicly known components are in gray.

by the pCloud CLI client does not perform any checks and even accepts an invalid key of the needed length (with $e = d = 1$).

For all the previously described attacks to succeed in practice, the adversary needs to sidestep client-side caching of the authentic key material, as otherwise the client will encounter decryption errors. For example, the server can replace the key material at registration or during folder creation, so that there will be no data encrypted with the authentic keys.

3.1.2 Sync and Tresorit: Unauthenticated Public Keys. We now discuss two attacks related to the lack of authentication of public keys of other users. This is a difficult problem to solve, as it usually requires a public-key infrastructure (PKI), out-of-band verification or similar mechanisms. These are features that are seldom found in cloud storage systems. In fact, due to the lack of authentication of public keys in Sync, a malicious server can always replace public keys supplied to the client when permanently sharing folders. The client will encrypt the new share key $\tilde{K}_{\text{share}}$ under an attacker-controlled key, giving the server access to the shared material. Tresorit takes a more considerate approach to public key authentication, as they deploy certificates to provide authentication of keys. However, since the certificates are signed using Tresorit's own CA, an adversary with access to Tresorit's servers will be able to sign arbitrary certificates. Much like Sync, an adversary may then replace public share keys pk_{sh}. Additionally, it may also try to replace admin keys pk_{A}. In particular, replacing pk_{A} during registration would allow the adversary to gain complete control over a user account. However, in this specific case, the application shows the fingerprint of the admin key, which mitigates this attack by allowing out-of-band verification.

3.1.3 Seafile: Protocol Downgrade Attack. Recall that Seafile supports multiple versions of its protocols and that the client uses *server-provided* information to choose which version to use. The server can then downgrade security to the one of the oldest version.

In our attack, the server targets the magic string generated by the client during the creation of a repository. The server downgrades the version used by the client to version 0, which uses the BytesToKey algorithm with an iteration count of 3. This means that the magic string generated will have, as its leading 20 bytes, the value $\text{SHA1}(\text{SHA1}(\text{SHA1}(\text{ID}_{\text{repo}} \,\|\, \mathcal{P})))$, where ID_{repo} is public. With modern commercial hardware and widely-available software such as Hashcat [24], it is possible to brute-force over 10^{10} SHA1 hashes per second [37], which severely endangers the passwords of users which use version 0 of the Seafile protocol.

3.1.4 Sync: Link Sharing. As described in Section 2.2.1, Sync users can share files by creating a share link, which contains a share password $\mathcal{P}_{\text{link}}$. However, $\mathcal{P}_{\text{link}}$ is embedded in the link as part of the path. Whenever the link is clicked, the password is sent to the server. This automatically violates the confidentiality of the file when it is shared with a link. Interestingly, this issue was known to Sync, as shown by the 2015 version of their cryptography whitepaper.[2] In the whitepaper, the developers explain how the share password is encoded as part of a URI fragment, which is never sent to the server. The motivations for this downgrade in security are opaque to us.

[2]This whitepaper has since been removed from the website and replaced with a different document.

3.2 Tampering with File Data

We present attacks that target the integrity of files. We exploit the use of unauthenticated cipher modes to modify plaintext for Icedrive and Seafile (Section 3.2.1). For Seafile and pCloud, we provide attacks that exploit incorrect or lacking authentication of the file chunks, allowing to remove or reorder chunks in a file (Section 3.2.2).

3.2.1 Icedrive and Seafile: Unauthenticated Encryption. Both Icedrive and Seafile use unauthenticated CBC mode for encrypting file content, which implies that neither files nor file names are integrity-protected. As usual with CBC mode, the attacker can violate integrity by changing the plaintexts in a semi-controlled manner: the content of any block can be arbitrarily flipped, at the cost of sacrificing the content of the block before, which will be replaced by a block of garbage. Seafile uses a fixed IV for the encryption of all chunks, which prevents the attacker from changing the content of their first block. Icedrive uses randomized IVs, but the IV is itself encrypted under a fixed IV, which does not allow the attacker to control it, preventing a change of the first block.

In Icedrive, the lax padding checks allow for an adversary to truncate files by removing the trailing blocks of ciphertext.

3.2.2 Seafile and pCloud: Unauthenticated Chunking. Both Seafile and pCloud support chunking of files. The encryption for each chunk is conducted separately, so the chunks are cryptographically independent. Thus, an authentication mechanism is required to ensure integrity of the file. In Seafile, no such mechanism is present, therefore a malicious server is able to reorder or remove chunks arbitrarily. In pCloud, the client uses a Merkle tree of HMAC tags to provide integrity, as described in 2.2.2. We show that this construction is insecure. Recall, first, that pCloud sends the entire Merkle tree to the server. Second, note that, by construction, any subtree is also a valid and authenticated Merkle tree. This means that the attacker, rather than serving the entire file to the client, can serve only a subtree, along with all sectors which are authenticated by that subtree, which the client will decrypt without raising errors. The consequence is that the server can decide to remove entire sectors from the user's data.

3.3 Tampering with Directory Structure and Metadata

We present a series of attacks that alter the directory structure for all providers under consideration. We present attacks against the binding between files contents, names and paths in Sync, pCloud, Icedrive and Seafile, enabling a server to exchange the names of two files and, for Icedrive, we showcase an attack that allows the server to truncate file names, exploiting the lack of authentication in the encryption (Section 3.3.1). Then, we exploit the fact that metadata is not integrity-protected in any provider, which allows an adversary to manipulate the metadata of uploaded files, such as time of creation, file type and size (Section 3.3.2).

3.3.1 Sync, pCloud, Icedrive, Seafile: Tampering with File Names and Locations. Two of the most important pieces of metadata information in cloud storage are, arguably, file names and file locations. Many of the providers we analysed do not authenticate the location of files, allowing an adversary to trivially move them within the storage. File names are often encrypted, but are rarely authenticated or bound to the file content itself. As the file name and location carry relevant semantic information about the file, this allows for a malicious server to mislead users on the contents of files, without tampering with them.

In Sync, files are uniquely identified by a server-chosen sync_id. Ideally, the data, name and path of the files should be bound to this identifier, so that the contents of two files cannot be swapped or moved. However, for Sync, no such binding is present, allowing the server to move files anywhere in the storage and to swap the names of files. In pCloud, file names are bound to their parent folder, as they are encrypted using the folder key K_{folder}, but they are not bound to the file contents. This means that (1) file names can be swapped within the same folder, and (2) file contents can be placed anywhere in the storage and associated to any existing file name. Note that the adversary cannot "move" an encrypted file name outside of its original folder, since it is encrypted with a folder-dependent key. However, the attacker can still apply our key replacement attack on folder keys (Section 3.1.1) to be able to inject new file names. In Icedrive, all files and file names are encrypted using the same key K_{master}, without any cryptographic mechanism to bind them together and to the file location. This allows the adversary to change location of files and swap file names. Furthermore, due to the usage of CBC mode, the adversary can truncate file names with the granularity of one block. Exploiting the malleability of CBC mode is possible, but hard in practice due to the requirement that file names should consist of only UTF-8 characters. Still, this must be seen as a flawed choice of primitive by the protocol designers. Seafile relinquishes complete control of the directory structure to the server, once again enabling the server to move files within a repository. File names are not encrypted nor authenticated and can be changed at the server's discretion.

3.3.2 All: Tampering with Additional Metadata. In all the providers we analyse, some metadata is unencrypted and unauthenticated and can be arbitrarily manipulated by the server. This metadata includes file size, type, and time of origin. Crucially, for shared files in Sync and Tresorit, it contains information about who created the file or folder. This means that the server can make it appear as if a file has been created by a different user, which is particularly relevant for shared folders. Additionally, the server can make it appear as if the user had shared a folder with any arbitrary user. For example, a malicious server could frame an employee by adding a company competitor to the list of people who have access to a confidential folder and accuse the employee of industrial espionage.

3.4 Targeted File Injection

We provide attacks against Sync and pCloud that allow a malicious server to place files in a user's directory. Specifically, the goal of the adversary is to insert a *chosen* file into the user's storage, in a way that is indistinguishable from a file that the user uploaded. As long as an injected file is indistinguishable from an honestly uploaded file, at least from the user interface, such an attack could be used to place incriminating material in the user's storage, allowing for blackmailing. In Sync, an attacker can inject entire folders into

the user's storage (Section 3.4.1). In pCloud, an adversary can add individual files in the root folder of the user (Section 3.4.2).

3.4.1 Sync: Folder Injection Attack. In Section 3.1.1, we have shown that a malicious Sync server can swap the user's share key with an adversary-chosen key, which also allows an attacker to inject arbitrary files in the storage. We now show an additional attack that can still inject a folder in the user's storage without needing key replacement.

The intuition is that the server can simulate the process of permanently sharing a folder, as if it originated from an honest user and was accepted by the victim. This requires the adversary to sample a new K_{share}, using it to encrypt the folder to be injected, and encrypt K_{share} with the user's public key. Whenever the user requests the contents of the root folder, the server includes the new shared folder. In some cases, this can already constitute an issue, since it makes it appear as if the user accepted a permanent share, possibly one which contains illegal or copyrighted material. To make the attack more severe, the server can also build upon the attack described in Section 3.3.2 to modify the metadata of the folder and make it appear in the user interface as if the user uploaded the folder themselves. This aesthetic change makes the injected folder indistinguishable from a honestly uploaded folder when observing the user interface. We note that the underlying code *will* treat it as a shared folder, but a human observing the client will not be able to notice such difference.

3.4.2 PCloud: File Insertion and Substitution. In pCloud, a malicious server is able to insert arbitrary files into the user's storage by using the fact that the user's public key pk is also known to the server. As all file keys in pCloud are encrypted using pk and are not authenticated, the server can generate new file keys $(K_{\text{file}}^{\text{enc}}, K_{\text{file}}^{\text{HMAC}})$, encrypt a chosen file with them and add the file to an arbitrary folder. Creating a valid encrypted file name proves to be harder, since the file names are encrypted using the folder key. However, we can build upon the observations made in Section 3.3.1 and use an encrypted file name belonging to a different file, exploiting the lack of binding between file contents and names. In fact, a direct consequence of the combination of these two attacks is that the pCloud server can substitute an existing file's contents with arbitrary data.

4 DISCUSSION

4.1 Common Failure Patterns

The analyses focused on MEGA and Nextcloud have highlighted severe issues in the design of those two specific systems. It is then natural to ask ourselves: *how widespread are these failure modes in the larger ecosystem?* Through the lens of our investigation, it is clear that these failure modes are not unique to MEGA or Nextcloud. Indeed, our attacks highlight a range of issues that are spread across the broader ecosystem of encrypted cloud storage. While we do recognise that there may be E2EE cloud storage providers outside of our analysis that do not suffer from these vulnerabilities, we also note that the services that we did analyse are major providers in the space. In fact, we observe how products that were developed *independently* happen to suffer from the same vulnerabilities.

We now highlight the common failure patterns that have affected the five providers in our work in the hope that these anti-patterns can act as advice for the practitioners who wish to develop secure E2EE cloud storage. We also provide general principles that can help with mitigation, though we refrain from giving the specifics. This is because each provider has specific engineering constraints, which might make any concrete advice inapplicable to them.

Misuse of Cryptographic Primitives. The first evident problem with many providers is the misuse of cryptographic primitives. The protocols we analysed often make use of primitives with insufficient security guarantees. For example, CBC mode provides IND-CPA security for Icedrive and Seafile but not integrity; RSA encryption with OEAP padding, a solid choice of primitive for public-key encryption, provides IND-CCA security but not authentication. In other cases, primitives are used incorrectly, as in Seafile and Icedrive, which reuse the same key-IV pair for multiple encryptions.

This problem can only be mitigated by gaining a better understanding of the required security properties. For the prior examples, a careful design process could have pointed out that file data should be encrypted using an authenticated encryption primitive and that RSA ciphertexts need to be authenticated. Such a process, however, requires (possibly expensive) cryptographic expertise to which many companies might not have access. At the same time, companies making strong marketing claims should be able to hold up to cryptanalysis as part of their responsibility towards clients.

Leaking Data and Metadata. The confidentiality of file contents is the primary concern of any E2EE cloud storage, and most providers tackle the problem by encrypting the data with a key unknown to the server. As discussed, this must be done using the correct primitives, lest we leak patterns in the file data, as in Seafile or Icedrive. However, this is insufficient: even when correctly using solid primitives to encrypt files, much information can be learnt from their metadata.

The former General Counsel for the NSA is quoted as saying: "Metadata absolutely tells you everything about somebody's life. If you have enough metadata, you don't really need content". Metadata has been the focus of many works in the literature [12, 13, 36, 41, 45], with proposals that aim to improve leakage resilience in cloud storage systems. Unfortunately, this concern has not spread to the broader developer community: all analysed providers leak at least the directory structure, if not the names of folders or files themselves. File types and file lengths, that many of the providers leak, can be often used to identify its contents [41]. It is clear that the current level of leakage leaves much to be desired.

A first mitigation is to encrypt all metadata and prevent the server from learning the structure of the user's storage. More specifically, the client should create a special file which contains the metadata of the entire storage, including the directory structure and the names of files and directories. This file can be stored encrypted on the server, so that it can be retrieved every time the user wants to access their files, and updated whenever a file or folder is created, updated or deleted. Files can then be associated with a directory-independent identifier, which does not reveal the directory structure to the server anymore. Due to access patterns, this may prove insufficient [12, 13, 36, 45], but all current mitigations for this problem have unsatisfactory overheads or strong assumptions (e.g. non-colluding servers) that often preclude their usage in large-scale systems. Still, not making metadata immediately available

to the server protects against snapshot adversaries and forces an attacker to persistently observe user interactions with the server in order to learn their patterns.

Lack of Integrity in the Storage. Even in cases where confidentiality is ensured for both data and metadata, providers must also guarantee their integrity. As a motivating example, Böek has shown that using CFB mode for encryption in OwnCloud could have allowed an attacker to inject a small amount of arbitrary code in an executable [10]. If the attacker knows the entire content of the executable, this also allows the injection of an arbitrary amount of malicious code. As another example, not checking the integrity of file names and paths could allow a malicious server to swap patients' medical data, leading to misdiagnoses. Further, an adversary that can inject arbitrary folders and files can mislead the user about the nature of the contents of their storage.

While this can happen in non-encrypted cloud storage as well, the fact that users manage their own cryptographic keys can be seen to imply that they have full control of the uploaded data. Indeed, this is the reasoning under which MEGA was designed to be end-to-end encrypted [20], since the provider could not be then held liable for copyrighted files uploaded by the users. In the presence of a vulnerability affecting storage integrity, this can imply two things. If the vulnerability is not publicly known, a malicious server can inject compromising material and report the user to the authorities, arguing that only the user could have uploaded it. On the other hand, if the vulnerability is known and the provider has deemed it a non-issue, then this allows for a variation of a *Trojan horse defence* where a criminal, who uses the storage to share illegal material, can claim that the files have been injected by the provider in order to frame them. Both of these results are, arguably, undesirable, which suggests that integrity cannot be treated as secondary with respect to confidentiality.

Mitigations must be applied at multiple levels. The usage of authenticated modes of encryption provides integrity at the most granular level. If files are chunked, the entire structure should be authenticated, for example, by using Merkle trees, preventing truncation and chunk reordering. Finally, file names and paths should be bound to the file contents. Since files are usually assigned unique IDs, an option is to include the ID as associated data when encrypting file data, metadata and paths with an AEAD scheme.

Missing Authentication for Public Keys. In the providers we have analysed, files are shared via out-of-band sharing of a symmetric key or by encrypting key material under the public key of the recipient. Obtaining authentic public keys is a well-known problem commonly resolved via a public key infrastructure (cfr. TLS) or by out-of-band communication (cfr. Signal's fingerprint comparison feature). We observe how none of the services that provide a sharing capability with public keys, namely Sync and Tresorit, provide an independent and secure mechanism to authenticate public keys. In particular, Sync provides no mechanism at all, while Tresorit provides an internal CA, which, under the assumption that Tresorit could be compromised, provides no security.

Showing a fingerprint to the user before sharing would enable detection of server misbehaviour without needing more profound changes to the cryptographic protocols, at the cost of a less intuitive user experience. Another more user-friendly alternative is to deploy a key transparency protocol, similar to the ones deployed by Whatsapp [30] and ProtonMail [46]. This would allow for public auditability of the mapping between users and public keys, preventing a server from serving arbitrary public keys to users.

4.2 Deploying Mitigations at Scale

Even after mitigations have been developed, deploying them remains an arduous task that must take into account backwards compatibility, especially considering native clients that need to be manually updated. Furthermore, re-encryption of files (e.g. when the provider wants to change the encryption scheme) can only be done in collaboration with the user. In other words, each user would have to download all their data, decrypt it, re-encrypt it under the new protocol, and upload it to the server again. Backendal et al. estimate that MEGA's 1000 petabytes of data would have required more than half a year at their peak bandwidth to re-encrypt [8], without accounting for the massive load on their infrastructure[3]. If complete mitigations were to be deployed for all the providers we analysed, we would expect similar hurdles.

Another approach is to opportunistically re-encrypt files as the user accesses them. In this case, the main issue would be preventing a malicious server from downgrading the protocol version by serving the original ciphertext, even after the file has been re-encrypted. This requires the client to keep track of the files that have already been re-encrypted and to decrypt those files exclusively under the new protocol. However, requiring the client to keep a long-term state is a very strong assumption, especially considering the prevalence of web-based clients.

Deploying mitigations is even more troublesome for self-hosted applications like Seafile, as not only the clients but also all server instances need to be updated. For Seafile, this makes the developers less inclined to update the protocol, as they believe it could negatively affect user experience [33].

4.3 Towards Secure E2EE Cloud Storage

The challenges in mitigation efforts emphasise the importance of having provable security guarantees as early as possible, to avoid the continuous cycle of discovering and then patching vulnerabilities. Such guarantees can only be provided if the company has access to cryptographic expertise, which is rare and expensive. This inevitably confines secure E2EE cloud storage to well-financed companies that have resources to hire cryptographers to design and analyse their protocols, raising a barrier to entry into the ecosystem.

Arguably, many of the providers in our investigation, as well as MEGA and NextCloud, did not have access to cryptographic expertise when developing their protocol. This hypothesis is corroborated by the usage of weak primitives and the existence of trivial vulnerabilities in their systems. The question then becomes: how can the cryptographic community better support companies, especially those which are less financially equipped, in developing cryptographic applications?

From our analysis, it's clear that *some* cryptographic know-how has reached the community of developers: almost always, passwords are hashed using an appropriate KDF prior to usage and sometimes solid primitives such as RSA-OAEP and AES-GCM are

[3] In fact, at the time of writing, MEGA has only deployed short-term remediations.

used. Nonetheless, we have seen that vulnerabilities appear when (solid) primitives are composed to obtain protocols. Paterson et al. [42] have argued for a "don't roll your own cryptographic protocol" mantra, when pre-existing designs can be used. However, for E2EE cloud storage, there are no standardised and open designs to be implemented. It should then become a collective task and exciting future work for the cryptographic community to create such a standard and to prove it secure. The resulting standard should be a collaboration between cryptographers, practitioners, and implementers, as advocated by Backendal et al. [7], in order to take into account the engineering constraints that exist in the real world.

5 CONCLUSIONS

We have conducted an analysis of the E2EE cloud storage landscape, focussing on five popular providers as a case study and investigating their cryptographic design. Our work unveiled ten classes of attacks against these providers targeting the confidentiality of files, the integrity of file data, file metadata, and the directory structure, as well as attacks allowing the injection of files into the user's directory. Our findings challenge the marketing claims made by some providers and, in several instances, compromise their E2EE guarantees. Nonetheless, not all protocols are created equal; Tresorit's design is mostly unaffected by our attacks due to a comparably more thoughtful design and an appropriate choice of cryptographic primitives. However, the absence of formal security analyses, the lack of freely-available source code, and the complex nature of the cryptographic design, present significant barriers to independent security evaluations.

We pointed out recurring patterns of cryptographic design failures replicated by different providers independently of each other, highlighting how non-trivial the challenges are in this setting, and how they require solid theoretical foundations to be tackled.

We concluded by providing possible mitigations for each of the anti-patterns. While our list of vulnerabilities and mitigations is by no means exhaustive, we believe that it can shed a light on the technical challenges faced by developers in the field.

The vulnerabilities pervading E2EE cloud storage highlight a critical blind spot in our grasp of the field. From an academic standpoint, cloud storage might seem like a largely resolved issue, marred only by a few problematic instances such as MEGA and NextCloud. However, our research demonstrates that the practical reality is quite different, and that the ecosystem is fundamentally flawed. Our findings strongly suggest that, in its current state, the ecosystem of E2EE cloud storage is largely broken and requires significant reevaluation of its foundations.

ACKNOWLEDGEMENTS

We thank Kenneth G. Paterson for helpful discussions. Jonas Hofmann is supported in parts by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

REFERENCES

[1] Tresorit AG. 2024. *End-to-End Encrypted Storage for Businesses | Tresorit.* https://tresorit.com/.
[2] Tresorit AG. 2024. *Tresorit - Encryption Whitepaper.* https://tresorit.com/resources.
[3] Martin R. Albrecht, Matilda Backendal, Daniele Coppola, and Kenneth G. Paterson. 2024. Share with Care: Breaking E2EE in Nextcloud. Cryptology ePrint Archive, Paper 2024/546. https://eprint.iacr.org/2024/546 https://eprint.iacr.org/2024/546.
[4] Martin R. Albrecht, Miro Haller, Lenka Mareková, and Kenneth G. Paterson. 2023. Caveat Implementor! Key Recovery Attacks on MEGA. In *Advances in Cryptology – EUROCRYPT 2023, Part V (Lecture Notes in Computer Science, Vol. 14008)*, Carmit Hazay and Martijn Stam (Eds.). Springer, Heidelberg, Germany, Lyon, France, 190–218. https://doi.org/10.1007/978-3-031-30589-4_7
[5] OpenSSL Project Authors. 2024. *EVP_BytesToKey.* https://www.openssl.org/docs/man3.1/man3/EVP_BytesToKey.html.
[6] Matilda Backendal, Hannah Davis, Felix Günther, Miro Haller, and Kenneth G. Paterson. 2024. A Formal Treatment of End-to-End Encrypted Cloud Storage. Springer-Verlag.
[7] Matilda Backendal, Miro Haller, and Kenny Paterson. 2024. End-to-End Encrypted Cloud Storage. *IEEE Security & Privacy* 22, 2 (2024), 69–74. https://doi.org/10.1109/MSEC.2024.3352788
[8] Matilda Backendal, Miro Haller, and Kenneth G. Paterson. 2023. MEGA: Malleable Encryption Goes Awry. In *2023 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 146–163. https://doi.org/10.1109/SP46215.2023.10179290
[9] Mihir Bellare, Anand Desai, Eric Jokipii, and Phillip Rogaway. 1997. A Concrete Security Treatment of Symmetric Encryption. In *38th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Miami Beach, Florida, 394–403. https://doi.org/10.1109/SFCS.1997.646128
[10] Hanno Böck. 2016. *Pwncloud – bad crypto in the Owncloud encryption module.* https://blog.hboeck.de/archives/880-Pwncloud-bad-crypto-in-the-Owncloud-encryption-module.html.
[11] Lara Bruseghini, Daniel Huigens, and Kenneth G. Paterson. 2022. Victory by KO: Attacking OpenPGP Using Key Overwriting. In *ACM CCS 2022: 29th Conference on Computer and Communications Security*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM Press, Los Angeles, CA, USA, 411–423. https://doi.org/10.1145/3548606.3559363
[12] Weikeng Chen, Thang Hoang, Jorge Guajardo, and Attila A. Yavuz. 2022. Titanium: A Metadata-Hiding File-Sharing System with Malicious Security. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/auto-draft-234/
[13] Weikeng Chen and Raluca Ada Popa. 2020. Metal: A Metadata-Hiding File-Sharing System. In *ISOC Network and Distributed System Security Symposium – NDSS 2020*. The Internet Society, San Diego, CA, USA.
[14] Larry Dignan. 2017. *Google plans to leverage G Drive for broader enterprise footprint, team management and collaboration.* https://www.zdnet.com/article/google-plans-to-leverage-g-drive-for-broader-enterprise-footprint-team-management-and-collaboration/.
[15] Dropbox, Inc. [n. d.]. *FORM S-1 REGISTRATION STATEMENT.* https://www.sec.gov/Archives/edgar/data/1467623/000119312518055809/d451946ds1.htm#toc.
[16] Dropbox, Inc. 2023. *Third Quarter 2023 Financial Results.* https://investors.dropbox.com/static-files/f2b4e840-27cf-41ab-8fb9-ed4d529a68fa.
[17] Cezary Dubnicki, Krzysztof Lichota, Erik Kruus, and Cristian Ungureanu. 2026. Methods and systems for data management using multiple selection criteria . https://patents.google.com/patent/US7844581B2/en.
[18] ef4. [n. d.]. *Encrypted libraries leak lots of information.* https://github.com/haiwen/seafile/issues/350.
[19] Hal Finney, Lutz Donnerhacke, Jon Callas, Rodney L. Thayer, and Daphne Shaw. 2007. OpenPGP Message Format. RFC 4880. https://doi.org/10.17487/RFC4880
[20] Charles Graeber. 2024. *Megaupload Is Dead. Long Live Mega!* https://www.wired.com/2012/10/megaupload-mega/.
[21] haiwen. [n. d.]. *Security Questions.* https://manual.seafile.com/security/security_features/#encrypted-library.
[22] haiwen. 2024. *haiwen/seafile - High performance file syncing and sharing, with also Markdown WYSIWYG editing, Wiki, file label and other knowledge management features.* https://github.com/haiwen/seafile.
[23] Miro Haller. 2022. Cloud Storage Systems: From Bad Practice to Practical Attacks. (3 2022). https://www.research-collection.ethz.ch/bitstream/handle/20.500.11850/555337/Msc_thesis_Miro_Haller1.pdf.
[24] Hashcat. 2024. *hashcat - advanced password recovery.* https://hashcat.net/hashcat/.
[25] Daniel Huigens. 2023. *Source Code Transparency.* https://github.com/twiss/source-code-transparency/blob/main/explainer.md.
[26] Daniel Huigens, Mark Watson, and Ryan Sleevi. 2023. *Web Cryptography API.* https://w3c.github.io/webcrypto.
[27] ISO/IEC 8825-1:2021(en) 2021. *ASN.1 encoding rules - Part 1: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER).* Standard. International Organization for Standardization, Geneva, CH.
[28] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. 2016. Verena: End-to-End Integrity Protection for Web Applications. In *2016 IEEE*

Symposium on Security and Privacy. IEEE Computer Society Press, San Jose, CA, USA, 895–913. https://doi.org/10.1109/SP.2016.58
[29] Vlastimil Klima and Tomas Rosa. 2002. Attack on Private Signature Keys of the OpenPGP Format, PGP(TM) Programs and Other Applications Compatible with OpenPGP. Cryptology ePrint Archive, Report 2002/076. https://eprint.iacr.org/2002/076.
[30] Sean Lawlor and Kevin Lewi. 2024. *Deploying key transparency at WhatsApp*. https://engineering.fb.com/2023/04/13/security/whatsapp-key-transparency/.
[31] ID Cloud Services LTD. 2024. *Icedrive - Secure Encrypted Cloud Storage*. https://icedrive.net.
[32] ID Cloud Services LTD. 2024. *Icedrive - Secure Encrypted Cloud Storage*. https://icedrive.net/encrypted-cloud-storage.
[33] Jonathan Xu (Seafile Ltd.). 2024. Personal communication.
[34] Seafile Ltd. 2024. *About - Seafile*. https://www.seafile.com/en/about/.
[35] Seafile Ltd. 2024. *Seafile - Open Source File Sync and Share Software*. https://www.seafile.com/en/home/.
[36] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. 2015. Privacy and Access Control for Outsourced Personal Records. In *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Jose, CA, USA, 341–358. https://doi.org/10.1109/SP.2015.28
[37] Phoronix Media. 2024. *Hashcat Benchmark - OpenBenchmarking.org*. https://openbenchmarking.org/test/pts/hashcat&eval=306f31f896ee6afac758df6db7589b6a2a232723#metrics.
[38] MEGA. 2024. *MEGA*. https://mega.io/.
[39] Nextcloud GmbH. 2018. *Nextcloud grew customer base 7x, added over 6.6 million lines of code and doubled its team in 2017*. https://nextcloud.com/blog/nextcloud-grew-customer-base-7x-added-over-6-6-million-lines-of-code-and-doubled-its-team-in-2017/.
[40] Kevin "Kenny" Niehage. 2020. Cryptographic Vulnerabilities and Other Shortcomings of the Nextcloud Server Side Encryption as implemented by the Default Encryption Module. Cryptology ePrint Archive, Paper 2020/1439. https://eprint.iacr.org/2020/1439 https://eprint.iacr.org/2020/1439.
[41] Kirill Nikitin, Ludovic Barman, Wouter Lueks, Matthew Underwood, Jean-Pierre Hubaux, and Bryan Ford. 2019. Reducing Metadata Leakage from Encrypted Files and Communication with PURBs. *Proceedings on Privacy Enhancing Technologies* 2019, 4 (Oct. 2019), 6–33. https://doi.org/10.2478/popets-2019-0056
[42] Kenneth G. Paterson, Matteo Scarlata, and Kien Tuong Truong. 2023. Three Lessons From Threema: Analysis of a Secure Messenger. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 1289–1306. https://www.usenix.org/conference/usenixsecurity23/presentation/paterson
[43] pCloud International AG. 2024. *pCloud - Europe's most secure cloud storage*. https://www.pcloud.com/eu.
[44] pCloud International AG. 2024. *pClout - About us*. https://www.pcloud.com/company/about.html.
[45] Benny Pinkas and Tzachy Reinman. 2010. Oblivious RAM Revisited. In *Advances in Cryptology – CRYPTO 2010 (Lecture Notes in Computer Science, Vol. 6223)*, Tal Rabin (Ed.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 502–519. https://doi.org/10.1007/978-3-642-14623-7_27
[46] Proton AG. 2024. *What is Key Transparency? | Proton*. https://proton.me/support/key-transparency.
[47] Keegan Ryan and Nadia Heninger. 2022. Cryptanalyzing MEGA in Six Queries. Cryptology ePrint Archive, Report 2022/914. https://eprint.iacr.org/2022/914.
[48] Inc. Sync.com. 2024. *Sync | About Sync*. https://www.sync.com/about/.
[49] Inc. Sync.com. 2024. *Sync | Secure Cloud Storage, File Sharing and Document Collaboration*. https://www.sync.com/.
[50] Nikos Virvilis, Stelios Dritsas, and Dimitris Gritzalis. 2011. Secure Cloud Storage: Available Infrastructures and Architectures Review and Evaluation. In *Trust, Privacy and Security in Digital Business - 8th International Conference, TrustBus 2011, Toulouse, France, August 29 - September 2, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6863)*, Steven Furnell, Costas Lambrinoudakis, and Günther Pernul (Eds.). Springer, 74–85. https://doi.org/10.1007/978-3-642-22890-2_7

A PROVIDER FOLDER STRUCTURE

We provide additional information about the protocol design of each provider, namely how the files and folders are structured and addressed for each provider. This information complements the protocol descriptions in Section 2.

A.1 Sync

In Sync, every file or folder is associated with a sync_id, a unique identifier which allows the client to fetch them from the server. The server is required to know the entire file structure, as it needs to return a list of sync_ids whenever a client queries a directory. The structure is not cryptographically protected, which allows the server to control the file structure at will.

A.2 pCloud

pCloud uses server-chosen folder IDs and file IDs to identify resources. Each folder is associated with a list of all IDs of contained files and subfolders. Even though the names are encrypted, this means that the server has knowledge of the entire folder structure.

A.3 Icedrive

The directory structure of the storage is known to the server, though the names of files and folders are encrypted. The client can ask the server for the contents of any folder (through a unique ID), to which the server will reply with a list of files and subfolders. There is no mechanism for the client to check the veracity of the information given by the server, which means that the client trusts the server on the directory structure.

A.4 Seafile

The server has full control of the folder structure, as it sees all directories, file names, and the chunks which compose each file. Seafile does not provide a cryptographic mechanism to verify the integrity of the folder structure. In particular, we remark that file names are leaked to the server, an issue which is known to the developers and that they are not planning to remediate [18].

A.5 Tresorit

Tresorit file URLs always contain the encrypted name of the parent directory. The server is therefore aware of the file structure and also of the plaintext names of all Tresors, as these are not encrypted. The server can also infer some of the file types depending on the thumbnails the client fetches for the frontend, e.g. if only one file type is fetched after opening a folder. By storing metadata within the encrypted file of the parent directory, the location and name of a file is bound to the correct folder and cannot be tampered with.