A Novel Cryptographic Framework for Cloud File Systems and CryFS, a Provably-Secure Construction

Sebastian Messmer, Jochen Rill, Dirk Achenbach, Jörn Müller-Quade

A Novel Cryptographic Framework for Cloud File Systems and CryFS, a Provably-Secure Construction

Sebastian Messmer[1], Jochen Rill[2], Dirk Achenbach[2], and Jörn Müller-Quade[3]

[1] mail@smessmer.de
[2] FZI Forschungszentrum Informatik
{rill,achenbach}@fzi.de
[3] Karlsruhe Institute of Technology (KIT)
joern.mueller-quade@kit.edu

Abstract. Using the cloud to store data offers many advantages for businesses and individuals alike. The cloud storage provider, however, has to be trusted not to inspect or even modify the data they are entrusted with. Encrypting the data offers a remedy, but current solutions have various drawbacks. Providers which offer encrypted storage themselves cannot necessarily be trusted, since they have no open implementation. Existing encrypted file systems are not designed for usage in the cloud and do not hide metadata like file sizes or directory structure, do not provide integrity, or are prohibitively inefficient. Most have no formal proof of security. Our contribution is twofold. We first introduce a comprehensive formal model for the security and integrity of cloud file systems. Second, we present CryFS, a novel encrypted file system specifically designed for usage in the cloud. Our file system protects confidentiality and integrity (including metadata), even in presence of an actively malicious cloud provider. We give a proof of security for these properties. Our implementation is easy and transparent to use and offers performance comparable to other state-of-the-art file systems.

1 Introduction

In recent years, cloud computing has transformed from a trend to a serious competition for traditional on-premise solutions. Elastic cost models and the availability of virtually infinite resources present an alternative to offers of a preset volume. The more bandwidth is available to consumers, the more economically reasonable it is to replace an on-premise solution with a cloud solution. In the wake of the PRISM disclosures, it seems naïve to trust in the security of one's data in the cloud, however. The scientific challenge for security researchers is to solve this dilemma by finding solutions without sacrificing the economic benefits of cloud technology.

Cryptographic research offers methods that guarantee the confidentiality and integrity of data in the presence of an adversary. The principle of cryptographic *proof* eliminates trust requirements by highlighting precisely which guarantees

hold under which assumptions. A proof of security makes use of a formal model in formulating security properties. Cryptographic schemes can then also be expressed in the terms of the formal model. Formal proofs of security constructively establish how a scheme achieves a security property (under given assumptions). This is a significant difference to the "ad-hoc security" method of eliminating vulnerabilities from a scheme until one can no longer conceive of any more attacks.

Provably-secure schemes are rarely adopted in practice. The abstract computational models that form the basis of cryptographic frameworks don't usually facilitate a straightforward implementation. Also, the concept of efficiency in these models differs from practical efficiency notions, so that many asymptotically efficient schemes are rather inefficient in practice. In contrast, there are many practical solutions to security challenges. They are deployed widely, but seldomly lend themselves to a formal security analysis and are thus analysed in an "ad-hoc" fashion.

Returning to the cloud scenario from before, a particular case in this area of conflict is outsourced file system data. Encrypting snapshots of file systems (backups) as one single block is certainly a mastered task. To update a single file in a huge file system, one were to re-encrypt and re-upload the whole snapshot. It is a different challenge altogether to efficiently deduplicate and compress encrypted remote backups to conserve bandwidth and storage space. In a similar fashion, it is not immediately obvious how to allow fine-grained access to single files in a file system hierarchy while *provably* protecting metadata and at the same time conserving efficiency. Indeed, we are not aware of any efficient cryptographic cloud file system in literature.

1.1 Our Contribution

Our contribution is twofold. We first give a formal security model for encrypted file systems and cloud file systems in particular. Our model covers both integrity and confidentiality for chosen ciphertext attacks, as well as chosen plaintext attack scenarios. Our model is designed to be as generic as possible to be useful for analysing the security of other cloud file systems beyond the scope of this paper.

Second, we design and implement CryFS[1], a provably secure encrypted file system for the cloud which is easy to use and acts completely transparent to the user. In addition to hiding file contents, we also hide file metadata, like sizes and permission bits, and the directory structure. Our file system is designed to be used by multiple users. When used only by a single user, CryFS also protects the integrity of the file system in the sense that no malicious storage provider can change the file system (for example delete, undelete or roll back files) without being noticed. We achieve good network performance by keeping ciphertext data in small same-sized blocks, which are organised in a special tree data structure and are synchronised individually. Local changes only cause few blocks to be synchronised. We prove that our file system is secure in our security model. The

[1] `https://www.cryfs.org`

performance of our reference implementation is already comparable to other state-of-the-art encrypted file systems. It is open source and available on github[2].

1.2 Related Work

There are various commercial and free solutions for secure cloud storage. Providers like SpiderOak[3], tresorit[4] and boxcryptor[5] offer cloud storage space in combination with a proprietary client application to synchronise data. They claim that all data is encrypted on the client and stored securely on the servers. However, these services do not disclose the specification of their protocols. Thus, they presume a certain level of trust in their service that is not much different from trusting a popular cloud provider in the first place. Traditional encrypted file systems like EncFS[6], eCryptFs[7] and NCryptFS [12] are open and theoretically usable in a cloud setting, however, they lack important security features: By encrypting files individually, they protect the content but leave metadata like the directory structure unencrypted. Using this, an attacker can easily distinguish a music CD collection (which has about 20 files per directory, 3MB each) from a folder containing only documents. Other solutions like the now-discontinued TrueCrypt[8], VeraCrypt[9], and dm-crypt[10], hide the directory structure by encrypting the whole file system into one big container. However, these solutions cannot be used in a cloud setting efficiently, as changing one small file in the file system causes the whole container to be re-encrypted and thus to be re-uploaded.

What is more, none of the presented solutions have a formal proof of security. There has been research into how to model the security of file systems, however, most of this research is directed at disk encryption schemes. Damgård et al. [5] for example introduce a formalisation of encryption schemes for file systems that is based on the Universal Composability framework. However, there are many artefacts in their model which are not relevant in the cloud setting (e. g. they explicitly model physical and logical sectors). Their model also misses components on which our security is based (i.e. different states for client and server) and thus is not well suited for our setting. Kristian Gjøsteen [8] and more recently Khati et al. [11] both introduce a game-based security model, which, however, is also only suited for modeling full disk encryption.

Modeling the security of outsourced data in general has been mainly investigated in the context of *searchable encryption* and *proofs of data possession* (PDP), as well as *proofs of retrievability* (POR). For searchable encryption, there are many different security models (e. g. by Chase et. al [4], Goh [9] and others) which are specifically designed for the corresponding scheme and cannot easily be applied to other settings and schemes. In addition, keeping the queries private is an important goal in the context of searchable encryption and is thus almost always included in the security model. For cloud based file systems, this is not as

[2] `https://github.com/cryfs/cryfs`
[3] `https://spideroak.com`
[4] `http://tresorit.com`
[5] `http://www.boxcryptor.com`
[6] `http://www.arg0.net/#!encfs/c1awt`
[7] `http://www.ecryptfs.org`
[8] `http://truecrypt.sourceforge.net`
[9] `https://veracrypt.codeplex.com`
[10] `https://gitlab.com/cryptsetup/cryptsetup/wikis/DMCrypt`

important. Achenbach et al. [1] introduce a more general security framework for modeling the security of outsourcing schemes but their model does not consider integrity. However, our framework is in part inspired by their ideas. There is a rich body of work regarding outsourcing schemes and corresponding security models which provide proofs of data possession and retrievability (e. g. Zhang et al. [13], Erway et al. [7] and Cash et al. [3]). Similar to our goals, all these schemes provide integrity for outsourced data. However, their requirements are fundamentally different. The goal of a PDP scheme is for a cloud provider to be able to prove that he has all of the outsourced data and that he did not modify it maliciously without requiring the user to hold a copy of the data himself and without having to download it. This is very useful if the server performs computations on the outsourced data without interaction of the user and the user wants to verify if all the data is still correct. In our case however, the server is only used for storage and users interact with the data only locally. Thus, all integrity checks can be performed by the user on the data itself. In order to achieve these particular integrity guarantees, PDP schemes require design and performance trade offs, which are also reflected in their security models. This makes the schemes incomparable to our scheme and the security models hard to adapt to our case.

2 A Security Model for Cryptographic File Systems

In this chapter, we introduce a novel formal security model for cloud file systems which covers both security and integrity in a non-adaptive as well as in an adaptive setting. We first give security definitions in the chosen plaintext attack scenario and then show how to extend them to the chosen ciphertext attack scenario. Further, we show that chosen ciphertext security for file systems can be achieved by combining plaintext security and integrity. Note that throughout this work we use $\cdot$ to denote a free parameter, which can be chosen by the adversary.

2.1 Basic Definitions

In general, encrypted file systems use a symmetric encryption scheme as underlying primitive. We give a formal definition of such an encryption scheme.

Definition 1 (Symmetric Encryption Scheme). *A symmetric encryption scheme $\mathcal{E}$ is a tuple $\mathcal{E} := (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ with*

- $\mathsf{Gen} : 1^k \to \{0,1\}^k$ *is a PPT algorithm which given a security parameter k, outputs a key K.*
- $\mathsf{Enc} : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^m$ *is a probabilistic polynomial time (PPT) algorithm which given a key K and a plaintext outputs the corresponding ciphertext.*
- $\mathsf{Dec} : \{0,1\}^k \times \{0,1\}^m \to (\{\perp\} \cup \{0,1\}^n)$ *is a PPT algorithm which given a key K and a ciphertext outputs the corresponding plaintext. It outputs $\perp$ if K is wrong or the ciphertext was not valid.*

Security and integrity of these basic encryption schemes are modeled by the standard security notions *indistinguishability under chosen plaintext* (IND-CPA) and *integrity of ciphertexts* [2] (INT-CTXT) respectively.

Security Game 1 ($\mathsf{IND\text{-}CPA}^{\mathcal{A}}(k)$)
- The experiment chooses a key $K \leftarrow \mathsf{Gen}(1^k)$ and a random bit $b \leftarrow \{0,1\}$.
- The adversary is given oracle access to $\mathsf{LR}(K, m_0, m_1)$, which outputs an encryption of m_b under K, if $|m_0| = |m_1|$.
- $\mathcal{A}$ submits a guess b' for b.

The result of the experiment is 1, if $b' = b$, and 0 else.

Security Game 2 ($\mathsf{INT\text{-}CTXT}^{\mathcal{A}}(k)$)
- The experiment chooses a key $K \leftarrow \mathsf{Gen}(1^k)$.
- The adversary is given oracle access to $\mathsf{Enc}(K, \cdot)$.
- The adversary is given oracle access to $\mathsf{Dec}(K, \cdot)$.

The result of the experiment is 1, if for any Dec oracle query: $\mathsf{Dec}(K, c) \neq \perp$ and c was never *output* by the Enc oracle.

Note that there are several equivalent formalisations for IND-CPA security [10]. We use the formalisation with a *left-or-right oracle* to reduce the complexity of our proofs. If a classic encryption oracle is needed, we can simulate it easily by setting both inputs to LR to be equal.

We now give a formal definition of an encrypted file system. In general, a file system needs four algorithms: one for initialising the file system (like setting up data structures), one for updating the file system (like adding and removing files), one for decrypting the file system and one for generating the cryptographic keys. The file system, and all algorithms which interact with it, are stateful.

Definition 2 (Encrypted File System). *Let $\mathbb{F}$ be the set of plaintext file systems, $\mathbb{C}$ the set of ciphertext file systems, and $\mathbb{S}$ the set of client states. Let $\mathbb{K} = \{0,1\}^k$ be the set of keys and $\mathcal{E} = (\mathsf{Gen}', \mathsf{Enc}', \mathsf{Dec}')$ be a symmetric encryption scheme. An encrypted file system $\mathcal{C}$ is a tuple $\mathcal{C} := (\mathsf{Gen}, \mathsf{Init}, \mathsf{Update}, \mathsf{Dec}, \mathcal{E})$ with*
- $\mathsf{Gen} : \{1\}^k \to \mathbb{K}$ *is a PPT algorithm which generates a key K.*
- $\mathsf{Init} : \mathbb{K} \to \mathbb{C} \times \mathbb{S}$ *is a PPT algorithm which takes the key K and initialises an empty ciphertext file system C, and the client state s.*
- $\mathsf{Update} : \mathbb{K} \times \mathbb{C} \times \mathbb{F} \times \mathbb{S} \to (\{\perp\} \cup \mathbb{C}) \times \mathbb{S}$ *is a PPT algorithm used to update the file system. It is given the key K, an old ciphertext file system C, a new plaintext file system F and a client state s. It outputs $\perp$ if the decryption of C fails, else a new ciphertext file system C', and a new client state s'.*
- $\mathsf{Dec} : \mathbb{K} \times \mathbb{C} \times \mathbb{S} \to (\{\perp\} \cup \mathbb{F}) \times \mathbb{S}$ *is a PPT algorithm which is given a key K, a ciphertext file system C, and the client state s and outputs $\perp$ if the decryption fails, else the decrypted file system F, and a new client state s.*

2.2 Modelling Non-Adaptive Security

Traditionally, security against non-adaptive adversaries requires that an adversary cannot gain any information from a scheme which they did not observe or interact with before. In the case of file systems however, we additionally require that the adversary could have interacted with other encrypted file systems using the same key. We allow the adversary to create an arbitrary but constant number

of file systems, which are available before and after he chooses the challenge. Also, we do not require the client state to be kept secret. We allow the challenges to be restricted by a relation R_d (e. g. both file systems must store the same amount of data). This means that from looking at a freshly encrypted file system, an attacker cannot deduce any information even if he observed modifications on different file systems using the same key. In particular, this requires the file system to introduce measures to be secure under key reuse (e. g. a user encrypting two different file systems with the same password). We call this security notion *indistinguishability under non-adaptive chosen file system attacks* (IND-naCFA).

Security Game 3 ($\mathsf{IND\text{-}naCFA}^{\mathcal{A},R_d}(k)$)

- The experiment chooses a key $K \leftarrow \mathsf{Gen}(1^k)$ and a random bit $b \leftarrow \{0,1\}$.
- The adversary is given oracle access to $\mathsf{Init}(K)$. The j-th query returns a new ciphertext file system (C_j, s_j) using the same key, and the following oracle to interact with it:
  - $(C'_j, s'_j) \leftarrow \mathsf{Update}_j(K, C_j, \cdot, s_j)$. The game sets $(C_j, s_j) := (C'_j, s'_j)$.

  The number of Init queries is bounded by an adversary-chosen constant q_{Init}.
- The adversary outputs two file systems F^0 and F^1 with $(F^0, F^1) \in R_d$.
- The experiment generates $(C, s) \leftarrow \mathsf{Init}(K)$.
- The experiment computes $(C', s') \leftarrow \mathsf{Update}(K, C, F^b, s)$.
- $\mathcal{A}$ is given (C, s) and (C', s').
- $\mathcal{A}$ submits a guess b' for b.

The result of the experiment is 1 if $b' = b$, and 0 else.

Definition 3 (Nonadaptive Security). *A file system is* IND-naCFA *secure, if*

$$\forall \mathcal{A}, c \in \mathbb{N} \exists k_0 \in \mathbb{N} \forall k > k_0 : |\Pr[\mathsf{IND\text{-}naCFA}^{\mathcal{A},R_d}(k) = 1]| \leq \frac{1}{2} + k^{-c}$$

2.3 Modelling Adaptive Security

Intuitively, while IND-naCFA models security of a file system directly after creation, adaptive security models the security of a file system later in its life. To achieve this, we allow the adversary to choose a file system as challenge with which he already interacted. We then require that he cannot distinguish which of two modifications he chose is performed. Again, we allow to restrict the adversary's choice of challenge by a relation R_d. We call this security notion *indistinguishability under adaptive chosen file system attacks* and it is a direct extension of IND-naCFA.

Security Game 4 ($\mathsf{IND\text{-}aCFA}^{\mathcal{A},R_d}(k)$)

- The experiment chooses a key $K \leftarrow \mathsf{Gen}(1^k)$ and a random bit $b \leftarrow \{0,1\}$.
- The adversary is given oracle access to $\mathsf{Init}(K)$, which on the j-th query initialises $F_j = \perp$ (empty file system), returns a new ciphertext file system (C_j, s_j) using the same key and an oracle to interact with it.
  - $(C'_j, s'_j) \leftarrow \mathsf{Update}_j(K, C_j, \cdot, s_j)$. The game remembers the most recent input F_j and sets $(C_j, s_j) := (C'_j, s'_j)$.

  The number of Init queries is bounded by a constant q_{Init} chosen by the adversary.

– The adversary outputs j and two file systems F^0, F^1 with $(F_j, F^0, F^1) \in R_d$.
– The experiment computes $(C'_j, s'_j) \leftarrow \mathsf{Update}_j(K, C_j, F^b, s_j)$ and passes (C'_j, s'_j) to the adversary.
– $\mathcal{A}$ submits a guess b' for b.

The result of the experiment is 1 if $b' = b$ and 0 else.

Definition 4 (Adaptive security). *A file system is* IND-aCFA *secure, if*

$$\forall \mathcal{A}, c \in \mathbb{N} \exists k_0 \in \mathbb{N} \forall k > k_0 : |\Pr[\mathsf{IND\text{-}aCFA}^{\mathcal{A},R_d}(k) = 1]| \leq \frac{1}{2} + k^{-c}$$

2.4 Modelling Integrity

To provide integrity, a cloud file system must ensure that a malicious server cannot alter the file system in any way, even though the server can observe every modification made to this file system and to other file systems using the same key. In particular, a server must not be able to provide the client with old states of the file system. This results in the following security model, which we call *integrity of file systems.*

Security Game 5 ($\mathsf{INT\text{-}FS}^{\mathcal{A}}(k)$)
– The experiment chooses a key $K \leftarrow \mathsf{Gen}(1^k)$.
– The adversary is given oracle access to $\mathsf{Init}(K)$. The j-th query returns a new ciphertext file system (C_j, s_j) using the same key, and the following oracles to interact with it:
  • $(C'_j, s'_j) \leftarrow \mathsf{Update}_j(K, C_j, \cdot, s_j)$. The game sets $(C_j, s_j) := (C'_j, s'_j)$.
  • $(F, s'_j) \leftarrow \mathsf{Dec}_j(K, \cdot, s_j)$. The game sets $s_j := s'_j$ for the next query.

  The number of Init queries is bounded by an adversary-chosen constant q_{Init}.

The result of the experiment is 1 if for any of the decryption oracle queries $\mathsf{Dec}_j(K, C', s_j) \neq \perp$, $C_j \neq C'$.

Definition 5 (Integrity). *A file system is* INT-FS *secure, if*

$$\forall \mathcal{A}, c \in \mathbb{N} \exists k_0 \in \mathbb{N} \forall k > k_0 : |\Pr[\mathsf{INT\text{-}FS}^{\mathcal{A}}(k) = 1]| \leq k^{-c}$$

2.5 Security Against Chosen Ciphertext Attacks

Like IND-CCA security is an extension of IND-CPA security, we extend IND-naCFA to IND-naCCFA and IND-aCFA to IND-aCCFA. The security games are identical to their chosen plaintext counterparts, except that Init returns an additional decryption oracle $\mathsf{Dec}_j(K, \cdot, s_j)$, which is modeled like in the INT-FS game.

For basic encryption schemes, ciphertext security (IND-CCA) can be achieved by combining plaintext security (IND-CPA) with integrity (INT-CTXT) [2]. We show that this is also true for file systems within our security framework.

Lemma 1. *A file system* $\mathcal{F} = (\mathsf{Gen}, \mathsf{Init}, \mathsf{Update}, \mathsf{Dec})$ *is* IND-(n)aCCFA *secure, if it is* IND-(n)aCFA *and* INT-FS *secure.*

Proof. Assume a modified version of IND-(n)aCCFA, where the Dec_j oracle only works for the most recent output of the corresponding Update_j oracle, or (if Update has not been called yet) for the output of Init. For all other queries, it returns $\perp$. We call this modified game IND-(n)aCCFA′. It is straightforward to reduce an adversary against IND-(n)aCCFA′ to an adversary against IND-(n)aCFA by remembering the most recent Update_j queries and answer the decryption query accordingly. We now show that any adversary with non-negligible success probability against IND-(n)aCCFA also has a non-negligible success probability against IND-(n)aCCFA′. Assume towards a contradiction an adversary A with a non-negligible different success probability in playing IND-(n)aCCFA and IND-(n)aCCFA′. We transform this adversary into an adversary A' against INT-FS. When A requests access to the Init oracle, A' forwards the calls to the Init oracle provided by INT-FS, returning (C_j, s_j) and the Update_j oracle. When A requests access to the Dec_j oracle, A' calls the Dec_j oracle provided by INT-FS, but ignores the response and implements the behaviour described for the IND-(n)aCCFA′ game by remembering the most recent Update_j query. For IND-aCCFA′, the challenge (C'_j, s'_j) is generated by another call to the Update_j oracle. For IND-naCCFA′, the challenge (C, s, C', s') is generated by calling Init and then using the freshly returned Update_j oracle. Since the success probability of A is non-negligibly different for IND-(n)aCCFA and IND-(n)aCCFA′, and the only difference in the games is the behaviour of Dec_j oracle queries that are not the most recent output of the Update_j oracle but decrypts successfully, we know such a query must happen with non-negligible probability. This query can be used directly to win the INT-FS game. □

3 CryFS: An Encrypted File System for the Cloud

CryFS is an overlay file system that can be mounted to a virtual folder. Everything the user stores in this virtual folder is encrypted in the background. The ciphertexts are stored on the hard disk (through the underlying file system) and can be picked up by third party synchronisation clients like Dropbox and uploaded to a cloud storage. This allows for a flexible use on top of any file system or cloud storage provider. In contrast to many other encrypted file systems, we do hide file contents as well as metadata like file sizes, file permissions and directory structure. We achieve this by splitting all file system data into same-size blocks. These blocks are then individually encrypted using an authenticated cipher. Using a specifically tailored data structure, we ensure that all file operations are still fast and we induce little space overhead, even though all files are segmented into small blocks (see Section 3.1). To prevent malicious storage providers from violating the integrity of the file system, we introduce additional measures to prevent rollback, deletion and re-introduction of deleted blocks (see Section 3.3). We point out that we decided against using hash trees to protect integrity: The primary reason behind this decision is our goal to support concurrent access to the file system. Hash trees induce changes from the affected block up to the root node, thus increasing the chance of edit conflicts. The second reason for

avoiding hash trees are performance considerations. Although hash trees have only logarithmic overhead in the size of the file system, any non-constant overhead is prohibitive for file systems with many frequent changes in many small files. Even though these integrity protections are only fully effective when the file system is used by a single user, CryFS is designed to work well with multiple users. See Appendix C for details. As most other encrypted file systems, CryFS uses two keys: a file system key for encrypting the file system blocks and a master key for encrypting the filesystem key. This makes it easy to change passwords for example.

3.1 Data Structures, Blocks and Files

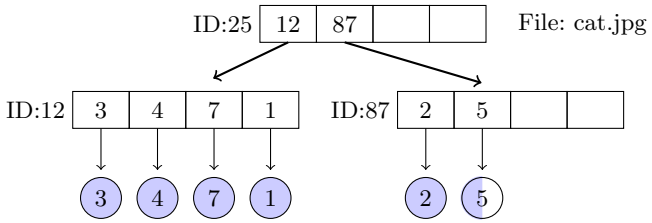

Fig. 1: The tree for an exemplary file "cat.jpg". Each tree node is one same-sized block in CryFS. The actual file data is stored in the leaves, whereas inner nodes store only pointers. For determining the file size, one only has to descend into the right-most branch of the tree and examine how much data is stored in the right-most leaf. Since all leaves are at the same depth and only the right-most elements are allowed to contain a less-than-maximum amount of data, this descend suffices to know how many blocks the file contains and thus the total file size.

As already mentioned, CryFS does not encrypt files individually. Rather, it splits every file into same-sized blocks, which are then encrypted. A tree data structure then associates blocks to files and files to directories. We base our construction on Dielissen et al.'s work on left-perfect binary trees [6] and generalise their definition to *left-max-data trees*.

The main idea for this data structure is that all nodes in the tree are as far left as possible. The actual binary file data is always stored in the left-most leaves of the file system tree and in-order. All leaves in the tree are at the same depth, and with exception of the right-most one, store exactly the same amount of data. This allows to represent arbitrary file sizes. Internal nodes contain only pointers to other blocks. If the block size is chosen appropriately (and thus the number of available pointers in each block), even large files can be represented by a tree with little depth. Every block is identified by a unique id, which is randomly chosen each time a block is created. See Figure 1 for an example file represented as a left-max-data tree. This structure leads to very efficient

algorithms for file system access. When trying to read a certain position in a file, one only needs to compute the respective block number from the total number of blocks in this file and the fixed block data size. Also, small changes to a file are particularly efficient: only a small block has to be changed (and synchronised to the cloud) not the whole file. Increasing the file size is described in Algorithm 1, decreasing is similar. Since only the right-most leaf can contain a less-than-maximum amount of data, determining the file size can also be achieved without reading all blocks by determining the amount of data in the right-most leaf. In our reference implementation with 32kb blocks and 16 byte block ids, this data structure induces a space overhead of roughly 0.05% for inner nodes plus an additive overhead of at most one leaf node's size if the right-most leaf is not full.

Algorithm 1 Grow an existing tree by one leaf

```
function GrowTree(treeRoot, newBlock)
    ℓ ← LowestNonFullInnerNode(treeRoot)
    if ℓ = ⊥ then /* All nodes are full. We need to add a level. */
        ℓ ← NewInnerNode() /* Create a new root block */
        ℓ.AddChild(treeRoot)
        treeRoot ← ℓ
    end if
    while depth(ℓ) < depth(leaves) − 1 do
        n ← NewInnerNode()
        ℓ.AddChild(n)
        ℓ ← n
    end while
    ℓ.AddChild(NewLeaf(newBlock))
    return treeRoot
end function
```

3.2 Directory Structure

Directories in CryFS are basically files themselves. Directories, however, do not store binary data but store a list of the directory's entries—i. e. pointers to the root block of files and directories. To allow for an efficient listing of all directory entries without having to descend into all individual file trees, we store the name of each entry, as well as all file metadata (like permission bits) along with the corresponding pointer in the directory structure. This layout allows for fast modifications of the directory structure. Moving a large directory only requires re-encrypting both the old and the new parent directory. See Figure 2 for an example of a file system tree with one directory and one file.

3.3 Encryption and Integrity

Encryption is on the block level—i. e. each block is encrypted individually. This allows for good performance because blocks can be encrypted in parallel. We use a cipher with an authenticated operation mode (e. g. AES-GCM) to prevent an adversary from altering the content of the blocks themselves. However, this is not yet sufficient to protect the integrity of the file system as a whole, since the

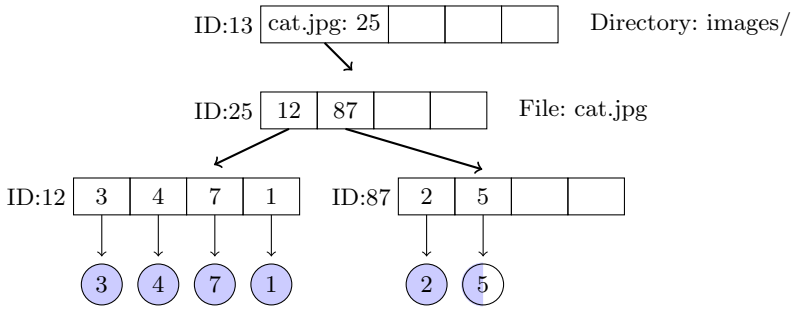

Fig. 2: The file "cat.jpg" is contained in a directory "images". To list all files of a directory efficiently, the name of each file is included with the respective pointer. As it is the case with files, once the number of entries in a directory exceeds the size of one block, the directory itself is represented as a tree.

connections between different blocks are not protected. An adversary can still try to reorder blocks, replace newer blocks with older versions, delete or re-add already deleted blocks.

We use a number of different mechanisms to prevent these attacks. First, we store the block ID in the header of the block, where it is integrity-protected by the authenticated encryption scheme. This ensures that an attacker cannot assign a different ID to a block (by changing the name of the file storing the block) and therefore prevents reordering. To prevent an attacker from replacing a block with a previous version of the same block, a block also stores a version counter in its header. Clients store a local list of all known blocks with a flag whether the block still exists, and their corresponding version numbers and check that it does not decrease. This list is also used to prevent an attacker from deleting or re-adding already deleted blocks without the client noticing. Additionally, the clients remember the master-key-encrypted file system key to prevent an adversary from replacing the whole file system including the key. In Section 4, we formally prove that this approach achieves the desired security goals. See Algorithms 1 – 5 for a description of relevant file system algorithms in pseudo-code.

4 Proving the Security of CryFS

In this section, we prove the adaptive and non-adaptive security of CryFS and show that it also provides integrity. Further, we show that CryFS also achieves ciphertext indistinguishability. We first give a formal description of CryFS. To simplify notation, we represent the tree structure of CryFS as a set of node blocks.

Definition 6 (CryFS). *Let* $\mathbb{I}$ *be the space of block IDs,* $\mathbb{I} \times \{0,1\}^n$ *the set of plaintext blocks, and* $\mathbb{I} \times \{0,1\}^m$ *the set of ciphertext blocks.* $\mathsf{CryFS}^{\mathcal{E}_1,\mathcal{E}_2}$ *is an* encrypted file system $(\mathsf{Gen}, \mathsf{Init}, \mathsf{Update}, \mathsf{Dec})$ *with* $\mathcal{E}_1 = (\mathsf{Gen}_1, \mathsf{Enc}_1, \mathsf{Dec}_1)$ *and*

Algorithm 2 Returns a new block with a unique id and the version number set to 0

```
function CreateBlock
    i ← GenerateUniqueID()
    return (i, i||0)
end function
```

Algorithm 3 Add a file or a folder tree to a directory

```
function AddToDirectory(directory, newEntry)
    if RightmostLeaf(directory).IsFull() then
        GrowTree(directory, CreateBlock())
    end if
    RightmostLeaf(directory).AddData(newEntry)
end function
```

Algorithm 4 Creates a tree data structure from a file and returns the root node

```
function CreateFile(file)
    D := (d_0, ..., d_n) ← SplitData(file)
    t ← CreateBlock()
    t.AddData(d_0)
    for all other d_i ∈ D do
        b_i ← CreateBlock()
        b_i.AddData(d_i)
        t ← GrowTree(t, b_i)
    end for
    return t
end function
```

Algorithm 5 Creates the data structure for a complete file system

```
function CreateFileSystem(sourceFileSystemRoot)
    rootBlock ← CreateBlock()
    for all Directories dir in sourceFileSystemRoot do
        rootBlock.AddToDirectory(CreateFileSystem(dir))
    end for
    for all Files file in sourcFileSystemRoot do
        rootblock.AddToDirectory(CreateFile(file))
    end for
    return rootBlock
end function
```

$\mathcal{E}_2 = (\mathsf{Gen}_2, \mathsf{Enc}_2, \mathsf{Dec}_2)$. *The client state* $\mathbb{S} \subseteq 2^{\mathbb{I}\times\mathbb{N}\times\{0,1\}} \times \{0,1\}^{k'}$ *stores a set of all known blocks with their id* $i \in \mathbb{I}$*, current version* $v \in \mathbb{N}$ *and a flag whether the block still exists* (1) *or was deleted in the past* (0)*. The state also stores* $c_{\text{fs}} \in \{0,1\}^{k'}$*, an encrypted version of the file system key. For the sake of clarity of the exposition, we first define intermediate functions:*

- $\mathsf{Repr} : \mathbb{F} \to 2^{\mathbb{I}\times\{0,1\}^n}$*: Takes a plaintext file system and generates its representation as a set of plaintext blocks.*
- $\mathsf{EncBlock} : \mathbb{K} \times (\mathbb{I} \times \{0,1\}^n) \times \mathbb{N} \to (\mathbb{I} \times \{0,1\}^m)$*: Takes a key* K_{fs}*, a plaintext block* (i, b) *and a version number* $v \in \mathbb{N}$*. Prepends block ID and version number to the data and encrypts it. Outputs* (i, c) *with* $c := \mathsf{Enc}_2(K_{\text{fs}}, i||v||b)$*.*
- $\mathsf{DecBlock} : \mathbb{K} \times (\mathbb{I} \times \{0,1\}^m) \to \{\perp\} \cup [(\mathbb{I} \times \{0,1\}^m) \times \mathbb{N}]$*: Takes a key* K_{fs} *and a ciphertext block* (i, c)*. Decrypts it to* $i'||v||b := \mathsf{Dec}_2(K_{\text{fs}}, c)$*. If decryption fails or* $i \neq i'$*, returns* $\perp$*. Otherwise, returns the plaintext block* (i, b) *and the version number* v*.*

Now we define the functions forming an encrypted file system.

- $\mathsf{Gen}(1^k) \mapsto (K_{\text{master}})$ *: Uses* Gen_1 *to generate a master key* K_{master}*.*
- $\mathsf{Init}(K_{\text{master}}) \mapsto (C, s)$ *: Takes* K_{master} *and generates* $K_{\text{fs}} \leftarrow \mathsf{Gen}_2(1^k)$*. Encrypts it with the master key to* $c_{\text{fs}} = \mathsf{Enc}_1(K_{\text{master}}, K_{\text{fs}})$*. Computes* $B := \mathsf{Repr}(F) = \{(i_0, b_0), \ldots, (i_n, b_n)\}$*, a set of blocks representing an empty file system* F*.*
  Sets $C := (c_{\text{fs}}, \mathsf{EncBlock}(K_{\text{fs}}, (i_0, b_0), 0), \ldots, \mathsf{EncBlock}(K_{\text{fs}}, (i_n, b_n), 0))$ *and* $s := (\{(i_0, 0, 1), \ldots, (i_n, 0, 1)\}, c_{\text{fs}})$ *and outputs* (C, s)*.*
- $\mathsf{Dec}(K_{\text{master}}, C, s) \mapsto (F, s)$ *: Reads* c_{fs} *from* C *and compares it with the* c_{fs} *stored in* s*. If they differ, returns* $\perp$*. Otherwise, decrypts it to* $K_{\text{fs}} := \mathsf{Dec}_1(K_{\text{master}}, c_{\text{fs}})$*. Then, computes*

$D := \{((i', b), v) \mid ((i', b), v) = \mathsf{DecBlock}(K_{\mathrm{fs}}, (i, c)), (i, c) \in C\}$. *Outputs* $\perp$ *in the following cases:*

- Dec_1 *fails to decrypt* c_{fs} *(wrong key or an integrity violation).*
- DecBlock *fails to decrypt* c *(wrong key, an integrity violation, or* $i \neq i'$*).*
- *There is an* $((i, b), v) \in D$ *for which there is no* $(i, v', 1) \in s$
- *There is an* $((i, b), v) \in D$ *for which there is an* $(i, v', 1) \in s$ *with* $v < v'$
- *There is an* $(i, v, 1) \in s$ *for which there is no* $((i, b), v') \in D$

Otherwise, computes the plaintext file system $F := \mathsf{Repr}^{-1}(\{(i_0, b_0), \ldots, (i_n, b_n)\})$ *and outputs* (F, s)*. The client state is not changed.*

– $\mathsf{Update}(K_{\mathrm{master}}, C, F', s) \mapsto (C', s')$: *Decrypts the old file system state to* $F := \mathsf{Dec}(K_{\mathrm{master}}, C, s)$*. Then, reads* c_{fs} *from* C *and decrypts it to* K_{fs}*. If either decryption fails, returns* $\perp$*. Initializes* $s' := s$*. Compares* $\mathsf{Repr}(F)$ *and* $\mathsf{Repr}(F')$ *and does the following:*

- *For each block* $(i, b) \notin \mathsf{Repr}(F)$*,* $(i, b') \in \mathsf{Repr}(F')$*:*
  - *If* $(i, v, 0) \in s$*, replace it in* s' *with* $(i, v + 1, 1)$*. Else, add* $(i, 0, 1)$ *to* s'
  - *Note: if* $(i, v, 1) \in s$*,* Dec *would have failed above.*
- *For each block* $(i, b) \in \mathsf{Repr}(F)$*,* $(i, b') \in \mathsf{Repr}(F'), b \neq b'$
  - *Replace* $(i, v, 1)$ *in* s' *with* $(i, v' + 1, 1)$*, where* v' *is the version number returned from* DecBlock *on decryption.*
  - *Note:* $(i, v, 1) \in s \wedge v' \geq v$*, otherwise* Dec *would have failed above.*
- *For each block* $(i, b) \in \mathsf{Repr}(F)$*,* $(i, b') \notin \mathsf{Repr}(F')$
  - *Replace* $(i, v, 1)$ *with* $(i, v, 0)$ *in* s'*.*
  - *Note:* $(i, v, 1) \in s$ *otherwise* Dec *would have failed above.*

Then, encrypts F' *using* EncBlock *with updated version numbers and outputs the new ciphertext file system* C' *(including* c_{fs}*), and the modified state* s'*.*

We now show that CryFS exhibits non-adaptive security according to Definition 3. We set R_d to restrict the challenge file systems to be representable using the same number of blocks. Formally, this means

$$R_d = \{(F^0, F^1) \in \mathbb{F} \times \mathbb{F} : |\mathsf{Repr}(F^0)| = |\mathsf{Repr}(F^1)|\}$$

Theorem 1 (Nonadaptive Security of CryFS).
$\mathsf{CryFS}^{\mathcal{E}_1, \mathcal{E}_2} = (\mathsf{Gen}, \mathsf{Init}, \mathsf{Update}, \mathsf{Dec})$ *is* IND-naCFA *secure, if* $\mathcal{E}_1 = (\mathsf{Gen}_1, \mathsf{Enc}_1, \mathsf{Dec}_1)$ *and* $\mathcal{E}_2 = (\mathsf{Gen}_2, \mathsf{Enc}_2, \mathsf{Dec}_2)$ *are* IND-CPA *secure encryption schemes.*

Proof. We prove the claim by reduction using two steps. First, we modify IND-naCFA to IND-naCFA′ such that when the adversary gets the challenge $(C, s), (C', s')$, it does not contain an encryption of K_{fs}, but an encryption of 0s instead. We prove that an adversary which has a different advantage in IND-naCFA and IND-naCFA′ can be used to break the IND-CPA security of $\mathcal{E}_1$. Second, we give a reduction from IND-naCFA′ to the IND-CPA security of $\mathcal{E}_2$.

Consider the following modification to IND-naCFA: When the adversary expects the challenge (C', s'), replace the encrypted file system key $\mathsf{Enc}_1(K_{\mathrm{master}}, K_{\mathrm{fs}})$ in state and ciphertext with $\mathsf{Enc}_1(K_{\mathrm{master}}, 0)$. We call this modified game IND-naCFA′. Now, assume towards a contradiction an adversary A with a probability of success p against IND-naCFA and p' against IND-naCFA′, where $p = p' + d$ for a positive

non-negligible d. This adversary can be used to construct an adversary B with a non-negligible advantage of $\frac{d}{2}$ against the IND-CPA security of $\mathcal{E}_1$. The reduction works as follows: The IND-CPA game draws $K_{\text{master}} \leftarrow \mathsf{Gen}_1(1^k)$ and a random bit b. When A uses the Init oracle, B generates $K'_{\text{fs}} \leftarrow \mathsf{Gen}_2(1^k)$ and (C_j, s_j) using the algorithms described in Definition 6 and uses the encryption oracle of IND-CPA to generate c'_{fs} as an encryption of K'_{fs}. Since B knows K'_{fs} it can also build the Update_j oracle. When the adversary outputs F^0, F^1, B generates another independent $K_{\text{fs}} \leftarrow \mathsf{Gen}_2(1^k)$, and passes 0 and K_{fs} as challenge to the IND-CPA game. The game returns c_{fs}. When $b = 0$, this is an encryption of 0. When $b = 1$, this is an encryption of K_{fs}. B then draws a random bit a, and knowing K_{fs}, can build the challenge (C, s) and (C', s') as an encryption of F^a. It replaces the encrypted file system key in C, s, C' and s' with the c_{fs} and returns the result to A. If A outputs a, A wins and B outputs 1 to the IND-CPA game. If A loses, B outputs 0. For $b = 0$, this was a perfect simulation of the IND-naCFA′ game. B has success probability $\Pr[a \not\leftarrow A \mid b = 0] = 1 - p'$. For $b = 1$, this was a perfect simulation of the IND-naCFA game. B has success probability $\Pr[a \leftarrow A \mid b = 1] = p = p' + d$. Together, B has success probability $\Pr[b \leftarrow B] = \frac{1}{2}(1 - p') + \frac{1}{2}(p' + d) = \frac{1}{2} + \frac{d}{2}$. Since d is non-negligible, B has a non-negligible advantage in the IND-CPA game which is a contradiction.

Now, assume towards another contradiction that A' is a successful attacker on IND-naCFA′. We transform A' into a successful attacker B' on IND-CPA security of $\mathcal{E}_2$: The game draws K_{fs} and a random bit b. B' draws $K_{\text{master}} \leftarrow \mathsf{Gen}_1(1^k)$. When A' uses Init, B' generates a new K'_{fs}, encrypts it with K_{master}, and creates an empty ciphertext file system. Knowing K_{master}, the Update_j oracle can be implemented easily.

Upon receiving challenges F^0 and F^1 from A', B' first generates an empty file system, and encrypts it to (C, s) using the encryption oracle and prepending $c'_{\text{fs}} = \mathsf{Enc}_1(K_{\text{master}}, 0)$. Then, B' updates it with F^0 and F^1 respectively, and uses the LR-oracle provided by IND-CPA successively for each pair of blocks in $\mathsf{Repr}(F^0)$ and $\mathsf{Repr}(F^1)$. This is possible, since we require $(F^0, F^1) \in R_d$ (i. e. both have the same number of blocks), Repr can be implemented to choose the same block ids for F^0 and F^1, and all blocks are of the same size. B' remembers all encrypted blocks returned by the oracle, prepends c'_{fs} to get C', and passes it to A' together with a generated file system state s' in which all block ids in have version number 1.

This is a correct simulation of the IND-naCFA′ game. When A' submits a guess for b, B' forwards it and thus inherits its success probability. This is a contradiction to the assumption that $\mathcal{E}_2$ is IND-CPA-secure. □

Theorem 2 shows that CryFS is also adaptively secure according to Definition 4. Since block IDs are public and CryFS only re-encrypts blocks for which the plaintext changed (for performance reasons), we set R_d to restrict both challenge file systems add, delete or modify blocks with the same block IDs. Theorem 3 shows that CryFS exhibits integrity according to Definition 5.

Theorem 2 (Adaptive Security of CryFS).
$\mathsf{CryFS}^{\mathcal{E}_1,\mathcal{E}_2} = (\mathsf{Gen}, \mathsf{Init}, \mathsf{Update}, \mathsf{Dec})$ *is* IND-aCFA *secure, if* $\mathcal{E}_1 = (\mathsf{Gen}_1, \mathsf{Enc}_1, \mathsf{Dec}_1)$ *and* $\mathcal{E}_2 = (\mathsf{Gen}_2, \mathsf{Enc}_2, \mathsf{Dec}_2)$ *are* IND-CPA *secure encryption schemes.*

Theorem 3 (Integrity of CryFS).
$\mathsf{CryFS}^{\mathcal{E}_1,\mathcal{E}_2} = (\mathsf{Gen}, \mathsf{Init}, \mathsf{Update}, \mathsf{Dec})$ *is* INT-FS *secure, if* $\mathcal{E}_1$ *is* IND-CPA *and* $\mathcal{E}_2$ *is* INT-CTXT *secure.*

The proofs for Theorem 2 and 3 can be found in Appendix A and B.

Lastly, we show that CryFS can also be secure against chosen ciphertext attacks.

Theorem 4 (Chosen Ciphertext Attacks). $\mathsf{CryFS}^{\mathcal{E}_1,\mathcal{E}_2} = (\mathsf{Gen}, \mathsf{Init}, \mathsf{Update}, \mathsf{Dec})$ *is* IND-naCCFA *and* IND-aCCFA *secure, if* $\mathcal{E}_1 = (\mathsf{Gen}_1, \mathsf{Enc}_1, \mathsf{Dec}_1)$ *is an* IND-CPA *and* $\mathcal{E}_2 = (\mathsf{Gen}_2, \mathsf{Enc}_2, \mathsf{Dec}_2)$ *an* IND-CPA *and* INT-CTXT *secure encryption scheme.*

Proof. This follows directly from Theorem 1, Theorem 3 and Lemma 1. □

5 Performance

In this section, we present results of our performance evaluation for our reference implementation of CryFS. We tested various performance factors in comparison to other popular file systems. Even though our implementation is preliminary and still has potential for optimisation, our experiments show that our file system has performance comparable to existing encrypted file systems and is practical.

CryFS is implemented using C++ and can be compiled with either GCC or Clang. For cryptography, the Crypto++[11] library is used, but the code is written in a way that allows for easy switching to another library. We tested CryFS 0.10-m2, EncFS 1.8.1, TrueCrypt 7.1a, and VeraCrypt 1.19. CryFS was built with GCC 5.3.1 using optimization level Ofast. In all cases, the underlying file system was Ext4. For comparision we also tested the performance of Ext4 itself without using a cryptographic file system on top. CryFS was configured to use aes-256-gcm and run with a block size of 32kb. EncFS was also set to aes-256. For TrueCrypt and VeraCrypt, a container with 50 GB size was created, also using aes-256. We used a machine with Intel(R) Core(TM) i5-2500K CPU @ 3.30GHz QuadCore, 16GB DDR3-RAM on Ubuntu 16.10, Linux 4.8.0-49 x86_64. As hard-drive, a Samsung HD 204UI was used. The experiments were run using the benchmarking tool bonnie++ 1.03e[12]. To minimize the influence of cache effects, bonnie++ runs the read/write tests with a test file size that is twice the size of main memory (32GB in our case). For create/stat and delete tests, we used $16 * 1024$ files with 10KB each. Each experiment was run three times to ensure a low standard deviation, and we report the average value. The benchmark script is available online. [13]

[11] `https://www.cryptopp.com/` [12] `http://www.coker.com.au/bonnie++/`
[13] `https://github.com/cryfs/benchmark/tree/0.10-m2`

We found that writes by CryFS on HDDs are 15% slower than EncFS, while random seeks are faster by 45%. Read performance is slower by about a factor of three. All operations are still fast enough to be used in practice, however. CryFS uses less CPU time for all operations. Table 1 includes measurements for all tested file systems and shows the measured performance in detail.

			CryFS	EncFS	TrueCrypt	VeraCrypt	Plain Ext4
Sequential Output	bytewise	MB/s	40.5(35%)	39.3 (38%)	29.2 (26%)	28.1 (25%)	70.9 (64%)
	blockwise	MB/s	53.8 (3%)	63.2 (8%)	34.7 (3%)	35.1 (3%)	71.0 (5%)
Sequential Input	bytewise	MB/s	20.9(23%)	65.7 (52%)	66.1 (59%)	67.4 (61%)	64.8 (69%)
	blockwise	MB/s	23.7 (1%)	67.8 (3%)	68.5 (3%)	69.0 (3%)	66.4 (4%)
Rewrite	blockwise	MB/s	19.3 (3%)	28.9 (4%)	31.4 (4%)	31.4 (4%)	31.6 (3%)
Random Seeks		/s	79.4 (0%)	53.5 (0%)	111.5 (0%)	108.3 (0%)	155.9 (0%)
Random Create		/s	2701 (6%)	4208 (12%)	4071(99%)	4036(99%)	–
Random Delete		/s	4070 (4%)	24250(19%)	9424(99%)	9457(99%)	–

Table 1: Experimental results for file system operations using the bonnie++ 1.03e benchmark. Bonnie++ tests sequential read and write speed, both bytewise and blockwise, and of a Rewrite run, which iteratively loads a block from the file, modifies it, and writes it back. It tests the performance of random seeks, creations and deletions. In parentheses next to each value, the average CPU utilization is reported.

6 Conclusion and Future Work

In this work, we introduced a novel formal model for the security and integrity of cloud file systems. Our model is generic and designed to be applicable for a wide range of file systems. We also introduced CryFS, a novel encrypted file system specifically designed for the cloud. It has low communication and storage overhead. It ensures the confidentiality of the file system by hiding file contents as well as metadata like file sizes and directory structure. It ensures the integrity of the file system even against a malicious storage provider when used by a single user, but can also be used efficiently by multiple users when integrity is not important. We proved the security of CryFS in our new framework. Our benchmarks show that CryFS offers comparable performance to other state-of-the-art file systems even though our implementation is preliminary and has room for improvements. Our implementation is available on github.

Regarding our framework, there are a few open questions to be addressed in the future. First, even though we establish basic relations between our security notions, it remains open to show other relations or separations to get a better understanding of the requirements for secure cloud file systems. Second, we show that if a basic encryption primitive is IND-CPA and INT-CTXT secure, it can be used to construct a IND-CCFA secure file system. It remains an open question, if IND-CCA security (which is a weaker notion) would also be sufficient. Last, extending our formal model to a multi-user setting as well as extending CryFS itself to provide integrity for multiple users is left for future work.

References

1. Achenbach, D., Huber, M., Müller-Quade, J., Rill, J.: Closing the gap: A universal privacy framework for outsourced data. In: Cryptography and Information Security in the Balkans - Second International Conference, BalkanCryptSec 2015, Koper, Slovenia, September 3-4, 2015, Revised Selected Papers. pp. 134–151 (2015)
2. Bellare, M., Namprempre, C.: Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. Journal of Cryptology 21(4), 469–491 (2008)
3. Cash, D., Küpçü, A., Wichs, D.: Dynamic proofs of retrievability via oblivious ram. J. Cryptol. 30(1), 22–57 (Jan 2017)
4. Chase, M., Shen, E.: Substring-searchable symmetric encryption. Cryptology ePrint Archive, Report 2014/638 (2014), `http://eprint.iacr.org/2014/638`
5. Damgård, I., Dupont, K.: Universally composable disk encryption schemes. Cryptology ePrint Archive, Report 2005/333 (2005), `http://eprint.iacr.org/`
6. Dielissen, V.J., Kaldewaij, A.: A simple, efficient, and flexible implementation of flexible arrays, pp. 232–241. Springer Berlin Heidelberg, Berlin, Heidelberg (1995)
7. Erway, C., Küpçü, A., Papamanthou, C., Tamassia, R.: Dynamic provable data possession. In: Proceedings of the 16th ACM Conference on Computer and Communications Security. pp. 213–222. CCS '09, ACM, New York, NY, USA (2009)
8. Gjøsteen, K.: Computer Security – ESORICS 2005: 10th European Symposium on Research in Computer Security, Milan, Italy, September 12-14, 2005. Proceedings, chap. Security Notions for Disk Encryption, pp. 455–474. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
9. Goh, E.J.: Secure indexes. Cryptology ePrint Archive, Report 2003/216 (2003), `http://eprint.iacr.org/2003/216`
10. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. Chapman and Hall/CRC cryptography and network security (2008)
11. Khati, L., Mouha, N., Vergnaud, D.: Full Disk Encryption: Bridging Theory and Practice, pp. 241–257. Springer International Publishing, Cham (2017)
12. Wright, C.P., Martino, M.C., Zadok, E.: NCryptfs: A secure and convenient cryptographic file system. In: Proceedings of the 2003 USENIX Annual Technical Conference. pp. 197–210. San Antonio, TX (Jun 2003)
13. Zhang, Y., Blanton, M.: Efficient dynamic provable possession of remote data via update trees. Trans. Storage 12(2), 9:1–9:45 (Feb 2016)

A Adaptive Security of CryFS

Theorem 2 (Adaptive Security of CryFS).
$\mathsf{CryFS}^{\mathcal{E}_1,\mathcal{E}_2} = (\mathsf{Gen}, \mathsf{Init}, \mathsf{Update}, \mathsf{Dec})$ *is* IND-aCFA *secure, if* $\mathcal{E}_1 = (\mathsf{Gen}_1, \mathsf{Enc}_1, \mathsf{Dec}_1)$ *and* $\mathcal{E}_2 = (\mathsf{Gen}_2, \mathsf{Enc}_2, \mathsf{Dec}_2)$ *are* IND-CPA *secure encryption schemes.*

Proof. Consider the following modification to IND-naCFA: When the adversary queries Init or the Update_j oracles or expects output (C, s), instead of getting $\mathsf{Enc}_1(K_{\text{master}}, K_{\text{fs}})$ they instead get $\mathsf{Enc}_1(K_{\text{master}}, 0)$. Now, assume towards a contradiction an adversary A with a success probability of p against IND-aCFA and a success probability of p' against IND-aCFA′, where $p = p' + d$ for a positive non-negligible d. This adversary can be used to construct an adversary B with

a non-negligible advantage of $\frac{d}{2}$ which breaks the IND-CPA security of $\mathcal{E}_1$. The game draws $K_{\text{master}} \leftarrow \mathsf{Gen}_1(1^k)$ and a random bit b. When A uses the Init oracle, B generates a new file system key $K_{\text{fs}} \leftarrow \mathsf{Gen}_2(1^k)$ and uses the LR oracle of the IND-CPA game to get c_{fs} as either an encryption of 0 or of K_{fs}, depending on the value of b. Then it generates a new empty file system (C_j, s_j) but replaces the encryption of K_{fs} with c_{fs}. A expects access to an Update_j oracle which can be built by using K_{fs} to decrypt and encrypt blocks. Again, B replaces all encryptions of K_{fs} with c_{fs}. When the adversary outputs j, F^0, F^1, B draws a random bit a. It uses Update_j to build the challenge (C', s') as an encryption of F^a. If A outputs a (A wins), B outputs 1. If A loses, B outputs 0. For $b = 0$, this was a perfect simulation of the IND-aCFA′ game. B has success probability $\Pr[a \not\leftarrow A \mid b = 0] = 1 - p'$. For $b = 1$, this was a perfect simulation of the IND-aCFA game. B has success probability $\Pr[a \leftarrow A \mid b = 1] = p = p' + d$. Together, B has success probability $\Pr[b \leftarrow B] = \frac{1}{2}(1 - p') + \frac{1}{2}(p' + d) = \frac{1}{2} + \frac{d}{2}$. Since d is non-negligible, B has a non-negligible advantage against IND-CPA.

Now, assume towards another contradiction that A' is a successful attacker on IND-aCFA′. We transform A' into a successful attacker B' on the IND-CPA security of $\mathcal{E}_2$. Intuitively, B' selects a random file system created by A' and uses A' to break its security. Since the number of file systems is a fixed constant, this only reduces the success probability by a constant amount. The reduction works as follows. The game draws K_{fs} and a random bit b. B' draws $\mathsf{K}_{\text{master}} \leftarrow \mathsf{Gen}_1(1^k)$ and draws a random $j^* \leftarrow \{1, \ldots, q_{\mathsf{Init}}\}$. When A' uses Init for the j-th time and $j \neq j^*$, B' generates a new K'_{fs}, encrypts it with K_{master}, and creates an empty ciphertext file system. Knowing K_{master}, the Update_j oracle can easily be implemented. In every output, $\mathsf{Enc}_1(K_{\text{master}}, K_{\text{fs}})$ is replaced with an encryption of 0. When A' uses Init for the j^*-th time, B' generates a new empty file system by using the encryption oracle of the IND-CPA experiment to encrypt all blocks. Again, B' prepends $\mathsf{Enc}_1(K_{\text{master}}, 0)$. B' also saves the current plaintext file system F_j (which is empty). If A' uses their access to the Update_j-oracle, B' updates the saved plaintext according to the input to the oracle. It uses the encryption oracle to encrypt added or modified blocks and exchanges them in the saved ciphertext. B' updates the saved file system F_j and the state s_j. Upon receiving challenge j, F^0 and F^1 from A', B' updates the corresponding plaintext F_j for both F^0 and F^1 respectively and passes the added and modified blocks of $\mathsf{Repr}(F^0)$ and $\mathsf{Repr}(F^1)$ (when compared to $\mathsf{Repr}(F_j)$) to the LR oracle of the IND-CPA experiment. It now has an encryption of either the modified blocks in F^0 or in F^1. Since it is required that $(F_j, F^0, F^1) \in R_d$ (i. e. they add, remove, and modify blocks with the same ID), B' knows which ciphertext blocks it has to add, remove and replace with their new versions in order to generate the correct ciphertext file system, even though it does not know which change was selected by the experiment. B' prepends $\mathsf{Enc}_1(K_{\text{master}}, 0)$ to the generated ciphertext and passes it to A' along with the updated state. This is a correct simulation of the IND-aCFA′ game. When A' submits a guess for b, B' forwards it to the game and thus inherits its success probability. This is a contradiction to the assumption that $\mathcal{E}_2$ is IND-CPA secure. □

B Integrity of CryFS

Theorem 3 (Integrity of CryFS). $\mathsf{CryFS}^{\mathcal{E}_1,\mathcal{E}_2} = (\mathsf{Gen}, \mathsf{Init}, \mathsf{Update}, \mathsf{Dec})$ *is* INT-FS *secure, if* $\mathcal{E}_1$ *is* IND-CPA *and* $\mathcal{E}_2$ *is* INT-CTXT *secure.*

Proof. Again, we first change INT-FS to INT-FS′ by replacing $\mathsf{Enc}_1(K_{\text{master}}, K_{\text{fs}})$ with $\mathsf{Enc}_1(K_{\text{master}}, 0)$ in the output of all oracles. Assume towards a contradiction that an adversary A with success probability of p against INT-FS and success probability of p' against INT-FS′ exists (where $p = p' + d$ for a positive non-negligible d). This adversary can be used to construct an adversary B with an advantage of $\frac{d}{2}$ against the IND-CPA security of $\mathcal{E}_1$ by using the following reduction: When A uses Init, B generates $K_{\text{fs}} \leftarrow \mathsf{Gen}_2(1^k)$ and uses the LR oracle of the IND-CPA game to get c_{fs} as either an encryption of 0 or of K_{fs}. It generates (C_j, s_j) using K_{fs} but replaces the encrypted file system key with c_{fs}. B builds the Update_j and Dec_j oracles using K_{fs} to decrypt and encrypt blocks. Each output contains c_{fs} instead of the encrypted file system key. When Dec_j is used, B checks whether decryption was successful for $C \neq C'$, i. e. whether A was successful. If A was successful, B outputs 1, otherwise it outputs 0. If $b = 0$, this was a perfect simulation of the INT-FS′ game. B has success probability $\Pr[0 \leftarrow B \mid b = 0] = 1 - p'$. If $b = 1$, this was a perfect simulation of the INT-FS game. B has success probability $\Pr[1 \leftarrow B \mid b = 1] = p = p' + d$ Together, B has success probability $\Pr[b \leftarrow B] = \frac{1}{2}(1 - p') + \frac{1}{2}(p' + d) = \frac{1}{2} + \frac{d}{2}$. Since d is non-negligible, this is a non-negligible advantage for B against IND-CPA.

Now, assume towards another contradiction that A' is a successful attacker on INT-FS′. We give a reduction which transforms A' into a successful attacker B' on INT-CTXT. The game draws $K_{\text{fs}} \leftarrow \mathsf{Gen}_2(1^k)$ and B' draws $K_{\text{master}} \leftarrow \mathsf{Gen}_1(1^k)$. B' draws a random $j^* \leftarrow \{1, \ldots, q_{\mathsf{Init}}\}$. When A' uses Init for the j-th time with $j \neq j^*$, B' generates a new independent K'_{fs} and creates a new ciphertext file system with this key. Knowing K'_{fs}, implementing Update_j and Dec_j oracles is straightforward. In every output, $\mathsf{Enc}_1(K_{\text{master}}, K_{\text{fs}})$ gets replaced by $\mathsf{Enc}_1(K_{\text{master}}, 0)$. When A' uses Init for the j^*-th time, B' creates a new empty file system but uses the encryption oracle provided by INT-CTXT to encrypt all blocks. It also builds Update_j and Dec_j but uses the decryption and encryption oracles of the INT-CTXT game to decrypt and encrypt. Instead of prepending $\mathsf{Enc}_1(K_{\text{master}}, K_{\text{fs}})$, which B' does not know, it prepends $\mathsf{Enc}_1(K_{\text{master}}, 0)$.

Since A' is successful, there is an oracle query $\mathsf{Dec}_j(K, C', s_j)$ which decrypts successfully with $C_j \neq C'$. With non-negligible probability $\frac{1}{q_{\mathsf{Init}}}$, this happens for $j = j^*$, where B' implemented Init using the INT-CTXT experiment. C_j and C' have the same set of block IDs, otherwise $\mathsf{Dec}_j(K_{\text{master}}, C', s_j) = \perp$. So there has to be a block in C' which is different from the corresponding block in C_j, i. e. $\exists i, c_i, c'_i : (i, c_i) \in C_j, (i, c'_i) \in C', c_i \neq c'_i$. This block c'_i was input to the decryption oracle of the IND-CTXT game when decrypting C'. We argue that c'_i wins the INT-CTXT game. First note that INT-FS′ decrypts with $c_{\text{fs}} = \mathsf{Enc}_1(K_{\text{master}}, K_{\text{fs}})$ from the state, not with the $c'_{\text{fs}} = \mathsf{Enc}_1(K_{\text{master}}, 0)$ passed to the adversary. Therefore c'_i decrypts successfully with the key from the INT-CTXT experiment. We now have to argue that c'_i was never output

by the INT-CTXT encryption oracle. Recall that this oracle is only used for encrypting the output of the j-th query of the Init oracle and for the outputs of the Update_j oracle. Since C' decrypts successfully, we know that the plaintext $((i', b'_i), v'_i) := \mathsf{DecBlock}(K, (i, c'_i))$ has ID $i = i'$ and a version number $v'_i \geq v^s_i$ where v^s_i is the version number in the state. All previous $\mathrm{Update'}_j$ oracle queries for this block ID encrypted a block with version number $v_i \leq v^s_i$, and $v_i = v^s_i$ only for c_i where we know $c'_i \neq c_i$. So we know c'_i was not output of the Update_j oracle. If (i, c'_i) was in the j-th output of the Init oracle, then $v'_i = 0$. In this case, either block i was never modified, which is a contradiction to $c_i \neq c'_i$, or block i was modified, which means $v^s_i > 0$ and is a contradiction to successful decryption. Taking everything into account, we know that c'_i was never output by the INT-CTXT encryption oracle and thus wins the game. This is a contradiction to the assumed security of $\mathcal{E}_2$. □

C Achieving Multi-User-Compatibility

CryFS provides confidentiality, integrity and fast file system operations in a single-user context. However, the design presented so far does not work well when used by multiple users for multiple reasons. For example, we cannot distinguish whether an integrity violation was caused by an attacker rolling back a block, or by a second client synchronising modifications on top of an outdated version. We resolve these problems by introducing a number of measures, which ensure that CryFS can be used with multiple users without integrity guarantees while maintaining integrity in the single-user setting.

First, in addition to having a pointer from the directory block to the root of a file, we also add a pointer from each file root node back to the directory it belongs to. That is, the whole directory structure is stored twice, once bottom-up in these pointers and once top-down through the file system tree. Using this, we can recover from a race condition where two users both add a different file to the same directory by periodically scanning for "dangling" pointers and reintegrate the corresponding files into the directory block.

Second, we extend the header of each block to also contain a unique client ID of the client who last modified the block along with the version counter. Further, each client saves the newest version for every block ID and client combination, and remembers the last updating client. Now, when a client reads a block that still has the same client ID as in his local state, the version number is checked to be non-decreasing otherwise it has to be increasing.

Third, instead of explicitly flagging deleted blocks in the local state, we set their last updating client ID to $\perp$. This allows clients to reintroduce deleted blocks as long as they increase the version number. Last, we allow to disable the check for missing blocks since there is no mechanism for a client to communicate, that he deleted a block, which will cause other clients to think that an attacker has deleted it.