Karlsruhe Institute of Technology

CryFS: Design and Implementation of a Provably Secure Encrypted Cloud Filesystem

Master's thesis

Sebastian Messmer

Institute of Theoretical Informatics
Competence Center for Applied Security Technology (KASTEL)
Department of Informatics
Karlsruhe Institute of Technology

Supervisor:	Prof. Dr. Jörn Müller-Quade
Advisor:	Dipl.-Inform. Jochen Rill

Writing Time: April 16, 2015 – October 15, 2015

I hereby declare that this thesis is my own work, and that I have not used any sources and aids other than those stated in the thesis.

Karlsruhe, October 15, 2015

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 15. Oktober 2015

Abstract

Cloud storage as offered by Dropbox and others is increasingly important for companies and individuals alike. However, the most cited limiting factors are confidentiality and integrity risks. To the best of our knowledge, there are no solutions that are secure and also easy enough to be used for cloud storage, and no solution for which there have been security proofs published. We introduce CryFS, a transparent and easy to use cryptographic filesystem, designed to be used with third party cloud storage solutions. Filesystem data is split into same-size blocks to be encrypted individually. This ensures confidentiality of file contents, file metadata and the directory structure. Integrity is achieved by keeping additional data like block version counters. We prove confidentiality and integrity using game based security notions. Different design alternatives are discussed and we develop balanced left-max-data trees, a tree data structure used by CryFS, which we prove to have minimal space overhead and to allow fast filesystem operations. We also provide a CryFS implementation and show that the filesystem is fast, allowing it to be used in practice.

Abstract

Immer mehr Firmen und Privatpersonen speichern wichtige Daten in der Cloud. Die meistgenannten limitierenden Faktoren sind Vertraulichkeits- und Integritätsrisiken. Unserer Kenntnis nach gibt es keine Lösung, die sicher und gleichzeitig einfach genug ist, um für Cloudspeicher eingesetzt zu werden, und keine Lösung, für die Sicherheitsbeweise veröffentlicht wurden. In dieser Arbeit stellen wir CryFS vor, ein transparentes und leicht zu benutzendes kryptographisches Dateisystem, dazu entworfen, mit Drittsoftware für Cloudsynchronisation zusammenzuarbeiten. Die Daten des Dateisystems werden in Blöcke gleicher Größe aufgeteilt und die Blöcke getrennt verschlüsselt. Das garantiert Vertraulichkeit der Dateiinhalte, Dateimetadaten und der Verzeichnisstruktur. Integrität wird erreicht, indem zusätzliche Informationen wie Versionsnummern gespeichert werden. Wir beweisen Vertraulichkeit und Integrität mit spielbasierten Sicherheitsbegriffen. Verschiedene Entwurfsalternativen werden vorgestellt und wir entwickeln balancierte leftmax-data Bäume, die von CryFS verwendet werden, und wir beweisen dass diese minimalen Platzverbrauch haben und schnelle Dateisystemoperationen ermöglichen. Wir stellen außerdem eine CryFS Implementierung zur Verfügung und zeigen, dass das Dateisystem schnell ist, wodurch es in der Praxis eingesetzt werden kann.

Contents

1 Introduction **1**
- 1.1 Contribution . . . 2
- 1.2 Related Work . . . 2
- 1.3 Structure . . . 4

2 Datastructure Theory **5**
- 2.1 Basics . . . 6
- 2.2 Perfect Trees . . . 6
- 2.3 Left-Perfect Trees . . . 7
- 2.4 Max-Data Trees . . . 9
- 2.5 Left-Max-Data Trees . . . 11
- 2.6 Balanced Trees . . . 11
- 2.7 Min-Data Trees . . . 12
- 2.8 Space Overhead . . . 14
- 2.9 Random Access . . . 16
- 2.10 Resizing . . . 16
  - 2.10.1 Querying Size . . . 16
  - 2.10.2 Growing . . . 17
  - 2.10.3 Shrinking . . . 18

3 Cryptography Basics **21**
- 3.1 Security Definitions . . . 21
  - 3.1.1 Security Goals . . . 21
  - 3.1.2 Attacker Models . . . 23
  - 3.1.3 Integrity . . . 25
  - 3.1.4 Relations . . . 26
- 3.2 Block Ciphers . . . 27
- 3.3 Modes of Operation . . . 28
  - 3.3.1 Electronic Codebook Mode (ECB) . . . 28
  - 3.3.2 Cipher Block Chaining Mode (CBC) . . . 29
  - 3.3.3 Counter Mode (CTR) . . . 30
  - 3.3.4 Galois Counter Mode (GCM) . . . 30
- 3.4 Conclusion . . . 31

4 System Design **33**
- 4.1 General Idea . . . 33
- 4.2 Design Goals . . . 34
  - 4.2.1 Achieving the Goals . . . 35

4.3 Design Overview . . . 39
4.4 Encryption Layer . . . 41
4.4.1 Integrity . . . 41
4.5 Blobstore Layer . . . 43
4.6 Filesystem Layer: Storing Directory Structure . . . 44
4.6.1 Central Directory Structure . . . 44
4.6.2 Directory Blobs . . . 46
4.6.3 Real Directories . . . 49
4.6.4 Path Headers . . . 51
4.6.5 Parent Pointers . . . 53
4.6.6 Conclusion . . . 55

5 System Reference **57**
5.1 Config File . . . 57
5.2 Block Layout . . . 58
5.3 Blob Layout . . . 59

6 Implementation and Evaluation **61**
6.1 Software Architecture . . . 61
6.1.1 Blockstore . . . 61
6.1.2 Blobstore . . . 63
6.1.3 Filesystem . . . 65
6.2 Performance Evaluation . . . 66
6.2.1 Experiment Setup . . . 66
6.2.2 Read and Write Tests . . . 67
6.2.3 Seek, Create, Stat and Delete Tests . . . 67
6.2.4 Conclusion . . . 67

7 Security Analysis **71**
7.1 Model . . . 71
7.2 Attacker Restrictions . . . 73
7.2.1 Confidentiality . . . 73
7.2.2 Integrity . . . 74
7.3 Confidentiality . . . 75
7.4 Integrity . . . 80
7.5 Adaptive Security . . . 86
7.5.1 Goals . . . 86
7.5.2 Attacker Types . . . 87
7.5.3 Adapting CryFS . . . 87
7.5.4 Database Privacy . . . 88
7.5.4.1 Readonly Queries . . . 88
7.5.4.2 Write Queries . . . 89
7.5.5 Query Privacy . . . 90
7.5.6 Conclusion . . . 90
7.6 Summary . . . 90

8 Conclusion **91**

Bibliography **93**

1. Introduction

Cloud storage is becoming increasingly important. Keeping personal files in the cloud allows for easy synchronization between different devices like computers, tablets or smartphones, or even sharing them with friends. Providers like Dropbox, Google, Microsoft, Amazon and many more are offering cheap cloud storage including synchronization clients running on the client devices. Unfortunately, most of these solutions store the user data on the servers in plaintext, so the user has to trust the storage provider to keep the files confidential and unmodified. However, in the current post-PRISM world, individuals and companies get increasingly cautious about security and privacy.

In a 2014 survey by the European Union [SR14], over 20% of individuals used cloud storage solutions to save documents, pictures, music, videos or other files, while 35% were aware of cloud services but did not use them. Of this last group, 44% cited security or privacy concerns as reason for not making use of such services. In the young generation (16-24), the percentage of people using cloud storage solutions was even higher at 35%. This predicts cloud storage to become increasingly used by individuals, with security and privacy being major barriers.

Further, companies start to use this opportunity for backup and archive, but also for storing files safely and for synchronizing them between devices and team members. In a 2014 survey by the European Union among companies [GS14], 19% used cloud services. Of them, 53% used it to store files. About half of the companies were classified as "highly dependent" on these cloud services. For both, large and small/medium companies, the risk of a security breach scored as the highest limiting factor for their usage of cloud services with 57% and 38% respectively.

A key requirement for a solution to this is simplicity of use. A solution where users have to encrypt their data manually and then upload encrypted containers will not be widely accepted. Since Dropbox itself is very simple to use, users expect the same from a secure cloud storage solution. To be equally simple, a solution has to keep cryptography transparently in the background without influencing the user's workflow.

1.1 Contribution

In this thesis, we design and implement CryFS[1], a provably secure filesystem for cloud storage. The filesystem is easy to use. Each client offers direct access to virtual plaintext files at its mount location, where the user can work with them and ignore the encryption layer. In the background, these files are encrypted and the ciphertext is stored to the local disk, where cloud clients can access it for synchronization. CryFS hides file contents, and also file metadata and the directory structure are provably secure from being read by attackers. Ciphertext data is kept in small same-size blocks, which are individually synchronized. Local changes only cause few blocks to be resynchronized. The reference implementation is open source and available on github[2]. While the primary target is to work together with cloud storage tools like Dropbox, it can easily be modified to use other synchronization software like Rsync[3] or Unison[4], or even to work without a local copy of ciphertexts and directly store the blocks remotely, for example on NFS or Amazon S3.

We also describe the theory behind balanced left-max-data trees, a tree structure developed for CryFS to store its data, describe the algorithms for accessing these trees and prove their correctness. Furthermore, we prove that balanced left-max-data trees only have very little space overhead and allow fast filesystem operations. We show that CryFS can be used in practice by providing experiments comparing CryFS to other filesystems. We discuss different design options on how directory structure can be stored and explain our design choices. We provide an overview over the software architecture of the reference implementation and give a reference for future implementators. Using game based security, we prove confidentiality and integrity of file contents, file metadata and directory structure. We also take a brief look at adaptive security scenarios, where an attacker gets information about the access patterns of filesystem operations.

1.2 Related Work

We discuss alternative solutions as well as research relating to this thesis.

Alternatives

There are small cloud storage providers like SpiderOak[5] that keep the user data on their servers in encrypted form, but their client software is not open source, so users still need to trust them and since the implementation is not open, neither is the security proven.

There is EncFS[6], which is open source, offers client side encryption and can be combined with any cloud storage provider to upload the encrypted data only, but it lacks important security features. EncFS encrypts files individually, but leaves the directory structure and file metadata unencrypted. Using this metadata, it is quite simple for an attacker to distinguish a music CD collection that has about 20 files per directory with about 3MB each from a folder containing documents. Furthermore, it is quite easy to figure out whether there is a directory containing a certain set of known files, for example the content

[1] http://www.cryfs.org
[2] https://github.com/cryfs/cryfs
[3] https://rsync.samba.org/
[4] http://www.cis.upenn.edu/~bcpierce/unison/
[5] https://spideroak.com
[6] http://www.arg0.net/#!encfs/c1awt

of a known software distribution CD. EncFS uses MACs to protect file integrity, but the enabling flag is kept in a config file which is stored with the encrypted data in plaintext. An attacker can simply switch it off. Even when enabled, integrity only works on per-file level and does not prevent adding or deleting files or replacing them with old versions. A security audit [Hor14] showed some other serious flaws in the current implementation of EncFS. In the conclusion, they state that EncFS is not safe if an attacker gets two or more snapshots of the ciphertext at different times. This makes it unusable for secure cloud storage, because a cloud storage provider has access to many snapshots over time. Some providers like Dropbox even explicitly store different file versions over time to allow rolling back to previous versions. In this scenario, not only the storage provider itself, but also an attacker having access to an account gets access to many snapshots of the ciphertext at different times.

Another potential solution is eCryptFs[7]. Like EncFS, it encrypts files individually and does not hide directory structure. Furthermore, it does not work well with cloud synchronization clients, because it does not allow them to change the underlying encrypted files while the filesystem is mounted. If another device makes some changes and they are synchronized, the cloud synchronization client updates the encrypted files, which causes undefined behaviour of the eCryptFs implementation.

There are other software solutions like TrueCrypt[8], which is discontinued, its unofficial successor VeraCrypt[9], and dm-crypt[10], that hide directory structure by encrypting the whole filesystem into one big container, but these solutions are hard to combine with cloud storage providers because providers synchronize on a per-file basis and do not know about the contents of the container file. Changing a small file in the filesystem can cause the whole container to be resynchronized, which leads to bad performance. Furthermore, even if two clients modify entirely different folders, this causes a synchronization conflict and only one of the versions survives. Thus, these solutions are not usable for cloud storage.

There is a scientific publication about eCryptFS [Hal05], but it only describes the design and does not contain security notions or proofs. There are some works providing analysis [Mia10] and describing various attacks [Tea11] on TrueCrypt, but none of them contains security notions or proofs. To the best of our knowledge, there are no scientific publications for SpiderOak, EncFS or VeraCrypt, and for none of the alternatives listed in this section there have been security proofs published.

Research

Damgård et al. [DD05] introduce a formalization of disk encryption that is based on the Universal Composability framework. In this thesis, however, we follow the simpler game based approach for modeling security. Kristian Gjøsteen [Gjø05] introduces some game based notions for disk encryption. They define non-adaptive and adaptive chosen plaintext and chosen ciphertext attacks against semantic security, non-malleability, plaintext-integrity and ciphertext-integrity. They also describe what kind of real life attacker corresponds to each of these attacker definitions. They establish relations between the security notions and

[7] http://ecryptfs.org/
[8] http://truecrypt.sourceforge.net/
[9] https://veracrypt.codeplex.com/
[10] https://gitlab.com/cryptsetup/cryptsetup/wikis/DMCrypt

build some simple hard disk encryption schemes to fulfill different notions. The security notions used in this thesis are similar to the ones they introduced. We also use some of their construction ideas to avoid common pitfalls in designing a hard disk encryption scheme. Dirk Achenbach et al. [Ach+15] introduces security notions for adaptive security. These notions are the basis for a brief look at adaptive security properties of CryFS in this thesis.

Matt Blaze et al [Bla93] develop the cryptographic filesystem CFS for Unix. The work was published in 1993 and the technologies used, e.g. DES, are outdated and known to be insecure by now. They define the filesystem and implement it for practical use, but they do not provide security notions or proofs.

Olaf Burkhart et al. [BM95] introduce the concept of left-perfect binary trees. We generalize this definition to left-perfect k-ary trees and lead this to the definition of balanced left-max-data trees. These are the kind of trees used by CryFS to store data.

1.3 Structure

In Section 2, we introduce the data structures used by CryFS. The filesystem uses balanced left-max-data trees, a class of trees specifically developed for CryFS. This section also introduces some algorithms and proves that these trees only have very little space overhead and allow fast filesystem operations. In Section 3, we introduce cryptographic primitives like block ciphers and explain the basic security definitions that are needed in later sections. Section 4 describes the filesystem design and explains how confidentiality and integrity are achieved. It also discusses some alternative designs and explains our choices. Section 5 contains a reference, so for example the layout used to store block data on disk. This can be used to implement other software working with CryFS encrypted data. Section 6 describes some of the details of the reference implementation and how we implemented the filesystem to be fast and secure. It also contains experiments comparing the performance of CryFS against alternatives. Section 7 contains the proofs for confidentiality and integrity of the user data stored in the filesystem.

2. Datastructure Theory

A *block* is an amount of data with fixed size. A *binary large object* or *blob* is an arbitrarily sized and resizeable amount of data. As described before, CryFS stores the filesystem using blocks. That is, it needs a way to store a blob like for example a file, while only using fixed size blocks.

In this section, we introduce balanced left-max-data trees, a special kind of tree used to split a blob into blocks. Intuitively speaking, a balanced left-max-data tree is a tree where all leaves are on the same level, all leaves are as far left as possible, and as few inner nodes as possible are used. CryFS stores each node in a block. Leaves store actual data and internal nodes store only pointers to their children. They enforce leaf order and allow fast random access.

Balanced left-max-data trees are based on left-perfect k-ary trees, which we introduce as a generalization of left-perfect binary trees [BM95]. We introduce algorithms for resizing, querying the blob size and random access. We also introduce perfect trees, which cannot store more leaves without increasing tree depth, and correspondingly max-data trees, which cannot store more bytes of data without increasing tree depth. Max-data trees are important for the resizing algorithm to recognize whether the tree depth has to be increased in an operation growing the blob. Min-data trees are introduced for the resizing algorithm to recognize whether the tree depth can be decreased in an operation shrinking the blob.

The following sections formalize these concepts. We also prove properties of balanced left-max-data trees and use these properties to prove that the algorithms are correct, space overhead is little and random access is fast.

Variable names

Throughout this thesis, the following variable names are used:

k	Number of children an inner node can store
L	Number of bytes a leaf can store
n	Number of nodes in a tree
ℓ	Number of leaves in a tree
d	Depth of a tree (not including root node level, a one-node tree has depth 0)
N	Number of bytes stored in a tree

2.1 Basics

In this section, we introduce the basic tree terminology used throughout the thesis. All trees are ordered trees, i.e. the children of a node have indices. Let T_1 and T_2 be trees with the same nodes, namely a root R and one child of the root C. In T_1, C has a different index than in T_2. Then T_1 and T_2 are different trees.

Definition 2.1 (k-ary tree)
A k-ary tree is a rooted tree where each node has at most k children with $k \geq 2$.

Corollary 2.2
A k-ary tree of depth d has at most k^d leaves.

Definition 2.3 (induced subtree)
A subtree S induced by a node K in a tree T consists of K and the nodes in the subtrees induced by its children (if any). The edges of S are the edges of T whose endpoints are both in S. The node K is the root of S.

Definition 2.4 (rightmost child)
The rightmost child of a node is the child with the highest index that actually exists.

Definition 2.5 (right-border node)
A right-border node is a node on the path from the root down to the rightmost leaf (always taking the rightmost child on the way).

2.2 Perfect Trees

This section introduces perfect trees [Toa], a kind of tree having the maximal number of leaves given a maximal depth; see Figure 2.1. They also have the minimal number of inner nodes any tree with the same number of leaves can have. That is, when storing that number of leaves, a perfect tree has the least possible space overhead.

Definition 2.6 (full k-ary tree)
A full k-ary tree is a k-ary tree where each node has either exactly k children (internal node) or 0 children (leaf).

Definition 2.7 (perfect k-ary tree)
A perfect k-ary tree is a full k-ary tree where all leaves are at the same depth.

Lemma 2.8
A perfect k-ary tree of depth d has exactly $l := k^d$ leaves, which is the maximal number of leaves any k-ary tree of depth d can have. All leaves are on level d. All nodes with less depth are internal nodes and have exactly k children.

Proof All leaves are on the same level by Definition 2.7. So all nodes on levels closer to the root are internal nodes. The leaves are on level d. Since a perfect k-ary tree is a full k-ary tree, the internal nodes have exactly k children. This is, the entire tree structure is determined by the depth. Level 0 has $k^0 = 1$ node (the root node). If level i has k^i nodes, then each of these nodes has k children and level $i + 1$ has $k^i \cdot k = k^{i+1}$ nodes. So level d has k^d leaves. Corollary 2.2 implies that this is the maximal number of leaves any k-ary tree of depth d can have. ∎

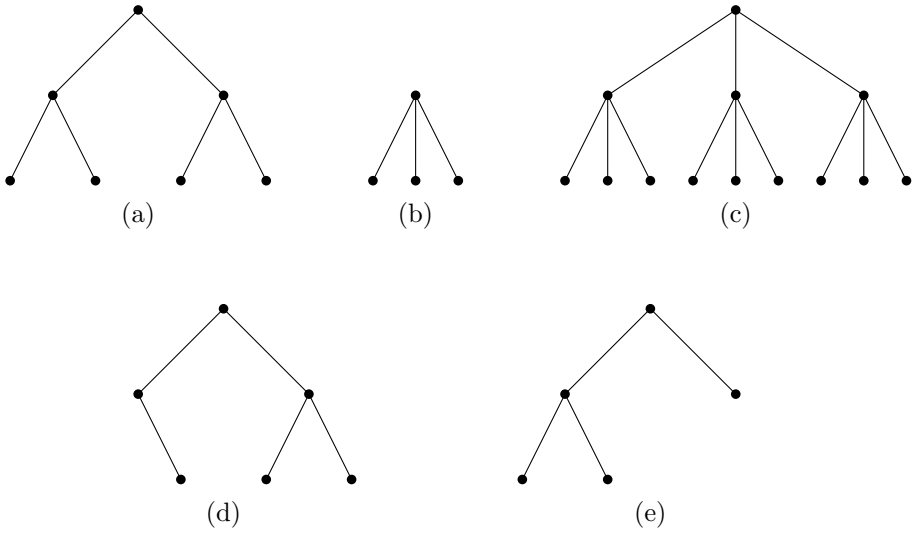


Figure 2.1 Examples for perfect and not-perfect k-ary trees. Tree (a) shows a perfect 2-ary tree and (b) and (c) show perfect 3-ary trees. Tree (d) has an internal node with less than k children, is therefore not full and not perfect. Tree (e) is a full tree, but not perfect, because not all leaves are on the same level.

Lemma 2.9
If a tree is a perfect k-ary tree, then all subtrees induced by its nodes are perfect k-ary trees.

Proof The nodes of the induced subtree are also nodes of the whole tree, so they have either exactly k or 0 children. This is, the subtree is a full k-ary tree. A subtree on level i has depth $d_i = d - i$. Since all leaves in the original tree are on level d, the leaves of this subtree are now on level $d - i$. So they are all on the same level. As a consequence, the subtree is a perfect k-ary tree. ∎

2.3 Left-Perfect Trees

As described before, a perfect tree stores a number of data leaves with minimal space overhead. However, perfect trees can only be used if the number of leaves is expressible as $\ell = k^d$; see Lemma 2.8. Most blobs have a number of leaves that is not expressible as $\ell = k^d$. To be able to reason about them, we introduce left-perfect k-ary trees.

The restriction remains that all leaves have to be on the same level. The reason for this restriction is that it simplifies the algorithms for calculating the blob size and randomly accessing a stored data byte with a certain index.

In their work about the Mathematics of Program Construction [BM95], Bernhard Möller et al. introduce a notion of left-perfect binary trees. In this section, we generalize this concept for k-ary trees; see Figure 2.2.

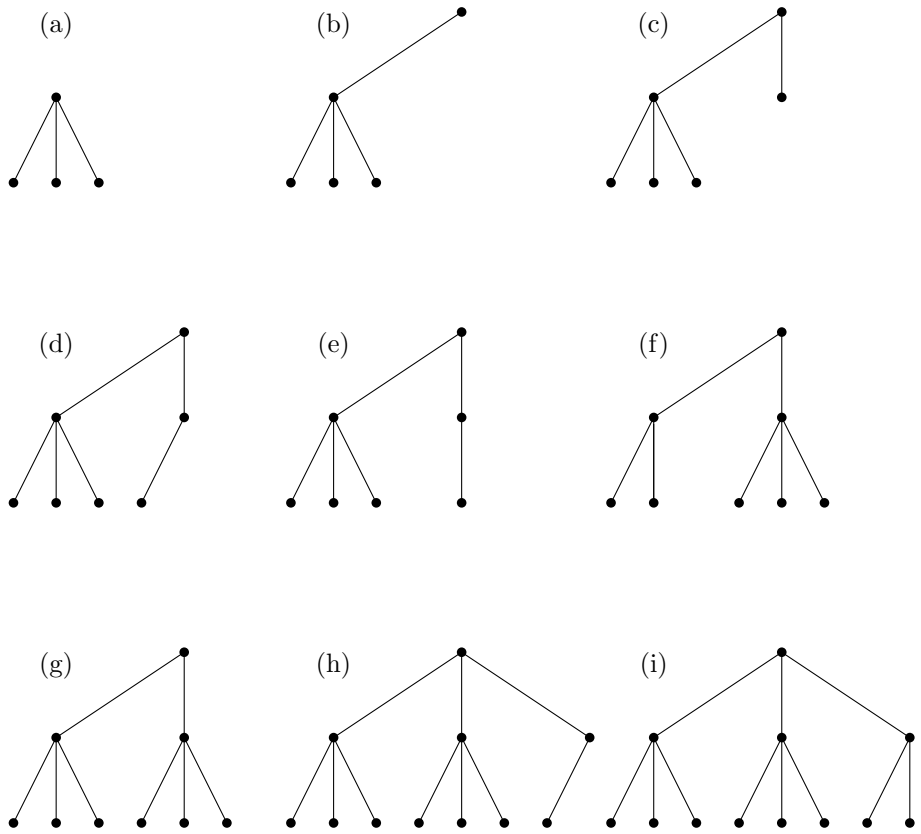


Figure 2.2 Examples for left-perfect and not-left-perfect k-ary trees. Tree (a) is a perfect tree and therefore also left-perfect. Tree (b) is a left-perfect 3-ary tree. Tree (c) is not, because not all leaves are at the same depth. Tree (d) is left-perfect. In tree (e) not all children are as far left as possible and in tree (f) the left child of the root does not induce a perfect tree, so they both are not left-perfect. Trees (g), (h) and (i) are left-perfect.

Definition 2.10 (left-perfect k-ary tree)
A left-perfect k-ary tree is a k-ary tree with the following properties:

1. *All induced subtrees except for the ones induced by the root or the rightmost child of a node are perfect k-ary trees. The root and rightmost children may but do not have to induce perfect k-ary trees.*
2. *All leaves are at the same depth.*
3. *All children of a node have the smallest possible indices, i.e. are as far left as possible.*

Lemmata 2.11 and 2.12 show a relation between left-perfect trees and perfect trees.

Lemma 2.11
A perfect k-ary tree is a left-perfect k-ary tree.

Proof In a perfect k-ary tree, all induced subtrees are perfect k-ary trees (Lemma 2.9) and all leaves are at the same depth. This is, the first two conditions for left-perfect k-ary trees are fulfilled.

In a perfect k-ary tree, each node has the maximum number of children. This way, all children have the smallest possible indices. The third condition for left-perfect k-ary trees is fulfilled. ∎

Lemma 2.12
In a left-perfect k-ary tree, all subtrees induced by non-right-border nodes are perfect k-ary trees.

Proof If a node is not a rightmost child, it has to induce a perfect k-ary tree by Definition 2.10. So the only nodes we have to look at are the nodes that are rightmost children but not right-border nodes. Let K be one of these nodes. Then there exists a node P on the path from K to the root, which is not a rightmost child, since otherwise, K is a right-border node. Since P is not a rightmost child, it has to induce a perfect k-ary subtree. Since the subtree induced by K is a subtree of the subtree induced by P, Lemma 2.9 implies that the subtree induced by K is also a perfect k-ary tree. ∎

Lemma 2.13
In a left-perfect k-ary tree, all subtrees induced by its nodes are left-perfect k-ary trees.

Proof Let K be a node in a left-perfect k-ary tree T. We now show that the subtree induced by K is a left-perfect k-ary tree.
Case 1: K is a non-right-border node. Then K induces a perfect k-ary tree by Lemma 2.12. As a consequence of Lemma 2.11, K induces a left-perfect k-ary tree.
Case 2: K is a leaf. In this case, it trivially induces a left-perfect k-ary tree.
Case 3: K is a right-border node, and not a leaf. For K to be a left-perfect k-ary tree, it has to fulfill Definition 2.10. Let us look at an arbitrary node P in the subtree induced by K and let C be a child of P. If C is not the rightmost child of P, it is not a right-border node of T and induces a perfect k-ary tree according to Lemma 2.12. This fulfills the first condition of Definition 2.10. All leaves of T are at the same depth, and all leaves of the subtree induced by K are also leaves of T. So they are also at the same depth. This fulfills the second condition. Since each node in the subtree induced by K is also a node in T, the third condition can also directly be derived from T being a left-perfect tree. ∎

2.4 Max-Data Trees

Using left-perfect k-ary trees, an arbitrary number of leaves can be stored with little space overhead. However, there still is the restriction that it can only store blob sizes that are a multiple of the number of bytes one leaf can store. To allow byte-grained resizeable blobs, it has to be allowed that a leaf stores less data than it actually could hold. This leads to the introduction of max-data trees in this section, and left-max-data trees in the next one. Max-data trees are storing the maximal amount of data any tree with its depth could hold. They are an important corner case for the resizing algorithms in Section 2.10.

Definition 2.14 (max-data leaf)
A max-data leaf is a leaf which stores the maximum number of bytes it can hold. It stores L bytes of data.

Definition 2.15 (max-data tree)
A left-perfect k-ary tree is called a max-data tree if it is holding the maximum number of bytes a left-perfect k-ary tree of its depth can possibly hold.

This is, if a max-data tree should be grown by one byte, the depth of the tree has to be increased.

Corollary 2.16
In a max-data tree, all leaves are max-data leaves.

The following Lemmata 2.17 and 2.18 explain the relation between perfect k-ary trees and max-data trees.

Lemma 2.17
A max-data tree is a perfect k-ary tree.

PROOF Assume T to be a tree which is a max-data tree but not a perfect k-ary tree. Let d be the depth of the tree. Since it is not a perfect k-ary tree, there either is an internal node with less than k children or there is a leaf not on level d; see Definition 2.7 and Definition 2.6.

Since T is a max-data tree, it is a left-perfect tree. By Definition 2.10, all leaves are on the same level. That is, to not be a perfect k-ary tree, T must have an internal node with less than k children.

If this is the case, then another child can be added to that node to store more data without changing the tree depth. Adding another child can be done without hurting the left-perfect property, so this operation is allowed.

This way, another byte of data can be stored without changing the tree depth. That is a contradiction to T being a max-data tree. ∎

Lemma 2.18
A perfect k-ary tree where all leaves are max-data leaves is a max-data tree.

PROOF Let T be a perfect k-ary tree where all leaves are max-data leaves. The two ways of adding more data to a tree are (a) adding another leaf and (b) adding more data to an existing leaf.

According to Lemma 2.8, another leaf cannot be added to the tree without changing the tree depth. Since each leaf is a max-data leaf, no leaf can store any more data. The tree T is a max-data tree. ∎

Corollary 2.19
Using Corollary 2.16 and Lemmata 2.17 and 2.18, the following statements are equivalent:

- *T is a max-data tree.*
- *T is a perfect k-ary tree where all leaves are max-data leaves.*

Lemma 2.20
A max-data k-ary tree with depth d where each leaf can store up to L bytes, stores $L \cdot k^d$ bytes.

PROOF A max-data tree is a perfect k-ary tree; see Lemma 2.17. According to Lemma 2.8, it has $\ell = k^d$ leaves. Since each leaf can store up to L bytes and is a max-data leaf, each leaf actually stores L bytes; see Corollary 2.16. The total number of bytes stored is $L \cdot \ell = L \cdot k^d$. ∎

2.5 Left-Max-Data Trees

Although max-data trees store a certain amount of data with minimal space overhead, this only works if the number of bytes is expressible as $L \cdot k^d$; see Lemma 2.20. This is often not the case. Very similar to the way we introduced left-perfect k-ary trees due to the size restrictions of perfect k-ary trees, we now introduce left-max-data trees because of the size restrictions of max-data trees.

Definition 2.21 (left-max-data tree)
A left-max-data tree is a left-perfect k-ary tree where all leaves except for the rightmost leaf are max-data leaves. The rightmost leaf is allowed to, but does not have to be a max-data leaf.

Lemmata 2.22 and 2.24 explain the relation between max-data trees and left-max-data trees.

Lemma 2.22
A max-data tree is a left-max-data tree.

Proof A max-data tree is a perfect k-ary tree by Lemma 2.17. So it is especially a left-perfect k-ary tree; see Lemma 2.11. In a max-data tree, all leaves are max-data leaves; see Corollary 2.16. This fulfills the definition of a left-max-data tree. ∎

Lemma 2.23
In a left-max-data tree, all nodes except for the right-border nodes induce max-data trees.

Proof Let T be a left-max-data tree and let P be a non-right-border node in T. By Definition 2.21, T is a left-perfect k-ary tree. P induces a perfect k-ary tree according to Lemma 2.12. Since P is not a right-border node, the rightmost leaf of T is not part of the subtree induced by P. So by Definition 2.21, all leaves contained in the subtree induced by P are max-data leaves. According to Lemma 2.18, the subtree induced by P is a max-data tree. ∎

Lemma 2.24
In a left-max-data tree, all nodes induce left-max-data trees.

Proof For non-right-border nodes, Lemma 2.23 together with Lemma 2.22 conclude the proof. Let us look at a right-border node P of a left-max-data tree T. According to Lemma 2.13, P induces a left-perfect k-ary tree. According to Definition 2.21, the only leaf in T which is allowed to be a non-max-data leaf is the rightmost one. This implies the same restriction for the subtree induced by P and fulfills Definition 2.21. ∎

2.6 Balanced Trees

The goal of using left-max-data trees is to use as little disk space as possible when storing a blob. Using a left-max-data tree already fulfills a lot of the preconditions for that, but does not handle one special case. It still allows a chain of single-child nodes at the root, therefore unnecessarily storing additional nodes, as shown in the examples in Figure 2.3. This leads to the definition of balanced left-max-data trees, which finally fulfill the goal.

Figure 2.3 Examples for trees that are left-perfect but not balanced, because the root node has only one child. In a balanced left-perfect tree, the root node is either a leaf or has at least two children.

Definition 2.25 (balanced left-perfect tree)
A balanced left-perfect tree is a left-perfect tree where the root is either a leaf or has at least two children.

Definition 2.26 (balanced left-max-data tree)
A balanced left-max-data tree is a balanced left-perfect tree which is also a left-max-data tree.

Lemma 2.27 shows that the problem in Figure 2.3 can only arise for left-perfect or left-max-data trees, but not for perfect or max-data trees.

Lemma 2.27
A perfect k-ary tree is a balanced left-perfect k-ary tree.

Proof According to Lemma 2.11, a perfect k-ary tree is a left-perfect k-ary tree. So the only thing left to prove is that it is balanced. If we assume it is not balanced, then the root has exactly one child; see Definition 2.25. Since the tree is a perfect tree, it is a full tree (see Definition 2.7), which means each node has either 0 or k children. This is a contradiction, because Definition 2.1 requires $k \geq 2$. ∎

Lemma 2.13 showed that any node in a left-perfect tree induces a left-perfect tree. Lemma 2.24 showed the same for left-max-data trees. With that, any node of a balanced left-perfect/left-max-data tree also induces a left-perfect/left-max-data tree, but the induced tree is not necessarily balanced. A simple example are right-border nodes in min-data trees from the following section.

Balanced left-max-data trees are the trees used by CryFS to store blobs using same-size blocks. The goal for using balanced left-max-data trees is to minimize space overhead. Later in Section 2.8, we discuss the space overhead for balanced left-max-data trees.

2.7 Min-Data Trees

Balanced left-max-data trees are used to store resizeable blobs with minimal space overhead and we introduced max-data trees to describe a situation where the number of stored bytes cannot be grown without increasing the depth of the tree. Analogously, we now introduce min-data trees to describe a situation where the depth of the tree can be decreased if the number of stored bytes is shrunk.

Definition 2.28 (min-data leaf)
A min-data leaf is a leaf which stores exactly one byte of data.

Min-data leaves are defined to contain one byte as opposed to zero bytes. The reason is that if a leaf contains zero bytes, it is not stored at all. So a min-data leaf contains the minimum amount of data while still being useful.

Definition 2.29 (min-data tree)
A left-max-data tree is called min-data tree if it is holding the minimum number of bytes that cannot be held in a left-perfect tree with smaller depth.

The following Lemma 2.30 explains how a min-data tree can be recognized.

Lemma 2.30
A left-max-data tree is a min-data tree if and only if the root is a leaf which stores only one byte or both the following conditions are met

1. *The root has exactly two children*
2. *The subtree induced by the right child of the root stores exactly one byte of data.*

Proof
Part 1: Let T be a left-max-data tree where the root is a leaf that stores only one byte. A tree with a lower depth could not hold any bytes, so this one byte is the minimum number of bytes that cannot be held in a left-perfect tree with a smaller depth. T is a min-data tree.

Part 2: Let T be a left-max-data tree of depth d which fulfills both conditions: The root has exactly two children and the subtree induced by the right child of the root stores exactly one byte of data. Let the left child of the root store i bytes of data. Then, the whole tree stores $i + 1$ bytes of data. In any left-max-data tree where the root has more than two children, the left child is a max-data tree and holds the maximal number of bytes any left-perfect k-ary tree of depth lower than d can hold; see Lemma 2.23 and Definition 2.15. That is, $i + 1$ is the minimum number of bytes that cannot be held in a left-perfect tree with a depth lower than d. T is a min-data tree.

Part 3: Let T be a left-max-data tree where the root is an inner node with only one child. Then, the subtree induced by this child is a tree with smaller depth holding the same number of bytes. T is not a min-data tree.

Part 4: Let T be a left-max-data tree where either the root has more than two children or the second child of the root holds more than one byte. In both cases, T can be transformed to a tree T' holding a smaller number of bytes that still cannot be held in a tree with smaller depth. If the root has more than two children, we can remove them and let T' be the tree consisting of the root node and the subtrees induced by the first two children of the root node. If the subtree induced by the second child of the root holds more than one byte, it can be replaced with a subtree that holds only one byte. In both cases, the number of bytes held in T' is larger than the number of bytes held in the subtree induced by its first child. Since its first child induces a max-data tree by Lemma 2.23, this first child holds the maximal number of bytes that can be held in a tree with depth $d - 1$; see Definition 2.15. That is, T is not a min-data tree. ∎

2.8 Space Overhead

Balanced left-max-data trees store a certain amount of data with little space overhead. In this section, we show that in a tree of depth d, they store at most d nodes more than the lower bound for k-ary trees. Furthermore, we show that when compared to only storing the data itself, the space overhead introduced by inner nodes can be approximated with $\frac{1}{k-1}$, which is 0.05% in the reference implementation.

The following Lemma 2.31 establishes a lower bound for the number of nodes any k-ary tree needs to store a certain number of leaves. Lemma 2.32 shows that perfect k-ary trees reach this lower bound.

Lemma 2.31
A k-ary tree T with ℓ leaves, the number of nodes is at least

$$n \geq \frac{k\ell - 1}{k - 1}$$

Proof When building a k-ary tree on top of ℓ leaves, the leaves can be seen as a forest F of ℓ trees. They are iteratively merged bottom-up until there is only one tree left in the forest. Each merge step adds the merged tree to F and removes at most k trees from F. That is, it shrinks the size of F by at most $k-1$. The final number of internal nodes is equal to the number of merge steps, because each such merge step introduces an internal node. That is, the number of merges should be minimal. This is achieved by having each step remove as many trees from F as possible, leaving fewer trees for future merge operations. So as long as F contains at least k trees, each step shrinks the size of F by $k-1$. In the beginning, the size of F is $a_0 := \ell$. After step i, the size of F is $a_i = a_{i-1} - (k-1)$. Resolving this recursive formula results in $a_i = \ell - (k-1) \cdot i$. This is done until $a_i \leq 1$, which happens for $i \geq \frac{\ell-1}{k-1}$. So in total, at least $\left\lceil \frac{\ell-1}{k-1} \right\rceil$ merge steps are needed, which means the tree has at least that many inner nodes. The lower bound for the total number of nodes is $n \geq \ell + \left\lceil \frac{\ell-1}{k-1} \right\rceil \geq \ell + \frac{\ell-1}{k-1} = \frac{k\ell-1}{k-1}$. ∎

Lemma 2.32
A perfect k-ary tree T with ℓ leaves has exactly $n = \frac{k\ell-1}{k-1}$ nodes.

Proof Let d be the depth of T. Level i has $n_i = k^i$ nodes, which means that there are $n = \sum_{i=0}^{d} k^i = \frac{1-k^{d+1}}{1-k}$ nodes in total. According to Lemma 2.8, the number of leaves is $\ell = k^d$. That is, $n = \frac{1-k\ell}{1-k} = \frac{k\ell-1}{k-1}$. ∎

Using Lemmata 2.31 and 2.32, a perfect k-ary tree storing ℓ leaves has the minimal number of nodes among all k-ary trees storing ℓ leaves. In the following, we establish an upper bound for left-max-data trees and compare this upper bound with the theoretical lower bound for k-ary trees.

Lemma 2.33
A balanced left-perfect k-ary tree of depth d with ℓ leaves has at most $n \leq \frac{k\ell-1}{k-1} + d$ nodes.

PROOF Let T be a balanced left-perfect k-ary tree of depth d with ℓ leaves. Let n_i be the number of nodes on level $d-i$. That is, n_0 is the number of nodes on the leaf level with $n_0 := \ell$. For any non-leaf level, all nodes except for the rightmost one have exactly k children; see Lemma 2.12. This implies the recursive formula

$$n_i = \left\lceil \frac{n_{i-1}}{k} \right\rceil \leq \frac{n_{i-1}}{k} + \frac{k-1}{k}$$

Solving this recursive formula results in

$$\begin{aligned} n_i &\leq \frac{\ell}{k^i} + \sum_{j=1}^{i} \frac{k-1}{k^j} = \frac{\ell}{k^i} + \frac{k-1}{k} \sum_{j=0}^{i-1} \frac{1}{k^j} = \frac{\ell}{k^i} + \frac{k-1}{k} \cdot \frac{1-\frac{1}{k^i}}{1-\frac{1}{k}} \\ &= \frac{\ell}{k^i} + \frac{(k-1)(1-\frac{1}{k^i})}{k-1} = \frac{\ell-1}{k^i} + 1 \end{aligned}$$

The total number of nodes is the sum over all levels

$$n = \sum_{i=0}^{d} n_i \leq \sum_{i=0}^{d} (\frac{\ell-1}{k^i} + 1) \leq d + 1 + (l-1) \sum_{i=0}^{d} \frac{1}{k^i} = d + 1 + (l-1) \frac{1-(\frac{1}{k})^{d+1}}{1-\frac{1}{k}}$$

Since T is a balanced left-perfect k-ary tree, the left child of the root induces a perfect k-ary tree. That is, the left child stores k^{d-1} leaves. The other children store at least one leaf or would not exist. So $\ell \geq k^{d-1} + 1$, which means $k^{d-1} \leq \ell - 1$ or $k^{d+1} \leq k^2(\ell - 1)$. Using that results in

$$n \leq d + 1 + (l-1) \frac{1-(\frac{1}{k^2(\ell-1)})}{1-\frac{1}{k}} = \frac{k\ell-1}{k-1} + d - \frac{1}{k^2-k} \leq \frac{k\ell-1}{k-1} + d$$

■

Lemma 2.33 shows that balanced left-max-data trees use at most d more nodes than the theoretical lower bound for k-ary trees. However, this theoretical lower bound for k-ary trees already uses inner nodes and therefore has more overhead compared to storing only the data itself. In the following, we show that this overhead is quite small.

The depth of the tree is $d = \lceil \log_k(\ell) \rceil$, which is a minor term in the formula of Lemma 2.33. For any practical purposes, d is a small constant with $d \leq 3$. Therefore, the space overhead can be estimated by using the formula $n \approx \frac{k\ell-1}{k-1}$. Calculating the relative space overhead results in

$$\frac{n-\ell}{\ell} \approx \frac{\frac{k\ell-1}{k-1} - \ell}{\ell} = \frac{\ell-1}{k\ell-\ell} = \frac{1-\frac{1}{\ell}}{k-1} \leq \frac{1}{k-1}$$

The higher k is chosen, the less overhead there is. However, since inner tree node have to be stored in a block, a higher k means a higher block size. Because of other performance factors, the block size should not be chosen too large. In our experiments, $k = 2048$ had the best runtime performance, which means there is a worst case overhead of

$$\frac{1}{2047} \approx 0.05\%$$

Because the blob size might not be divisible by the number of bytes one leaf can store, the last leaf has to be allowed to store less bytes than it could. This leads to an additional additive overhead of the size of one block.

2.9 Random Access

One goal for the datastructure is to have fast random accesses. This section describes how this is achieved. For a random access, the tree needs to be traversed from the root to the leaf containing the accessed byte. When following this path, properties of balanced left-max-data trees can be used to decide into which child to descend.

For that, each node K stores the depth of the subtree induced by K and the algorithm uses the following Lemma 2.34.

Lemma 2.34
Let T be a balanced left-max-data tree with depth d, which can store L bytes per leaf. Let $C_1, \ldots, C_n$ be the children of its root node. Then the subtrees induced by $C_1, \ldots, C_{n-1}$ store $L \cdot k^{d-1}$ bytes each.

Proof Lemma 2.23 implies that the subtrees induced by $C_1, \ldots, C_{n-1}$ are max-data trees. Their depth is $d' := d - 1$. According to Lemma 2.20, a max-data tree of depth d' stores $L \cdot k^{d'}$ bytes. ∎

Let T be a balanced left-max-data tree with depth d, which can store L bytes per leaf and N bytes in total. Let $C_1, \ldots, C_n$ be the children of its root node. Let the algorithm be run for a random access to the byte with index m. Then using Lemma 2.34, this byte is stored in the subtree induced by child C_i with

$$i := \left\lfloor \frac{m}{L \cdot k^{d-1}} \right\rfloor$$

Using this formula, the algorithm can follow the path from the root to the leaf containing the byte with index m by accessing d blocks. Since $d = \lceil \log_k \ell \rceil = \lceil \log_k \frac{N}{L} \rceil$, it has a theoretical random access time of $O(\log N)$. However, in practice, this access is very fast and can be seen as constant time. The reference implementation uses $k = 2048$ and $N = 32\text{KB}$. That is, for blobs below 64MB, $d = 1$ is enough. This is probably enough for most blobs stored in the filesystem. With $d = 2$, a 128GB blob can be stored and with $d = 3$ up to 256TB.

2.10 Resizing

Another goal is to be fast when resizing a blob. This section explains how this goal is met. We describe how the size of a blob can be queried and introduce Algorithm 1 for growing and Algorithm 2 for shrinking a blob. In Lemmata 2.35 and 2.36, we prove their correctness.

2.10.1 Querying Size

First, we explain the way the size of a blob can be queried. Very similar to the way random access works, this can also be done recursively. Let T be a balanced left-max-data tree with depth d, which can store L bytes per leaf and N bytes in total. Let $C_1, \ldots, C_n$ be the children of its root node. Then, $C_1, \ldots, C_{n-1}$ store $L \cdot k^{d-1}$ bytes each and the algorithm recursively descends into C_n to get the number of bytes stored there. Like the random access algorithm, this is done in $O(\log N)$ time.

If a faster way to get the blob size is needed, the blob size could be stored in the root node and updated on each resize. This would however have the disadvantage that the root node is changed on each resize. If an attacker can coerce the user to resize a blob, they can figure out which block is the root block of this blob. Not storing the blob size in the root node and using the recursive algorithm is therefore more secure.

2.10.2 Growing

Algorithm 1 shows how to grow a balanced left-max-data tree T by one byte. It can be extended to grow a tree by multiple bytes efficiently. The algorithm handles three cases. Either the last leaf can store an additional byte (case 1), or a leaf has to be added to the

```
 1: R ←root node of T
 2: L ← R, P ← ⊥, d ← 0, d_P ← 0
 3: while L has children do
 4:     if L has less than k children then
 5:         P ← L
 6:         d_P ← d
 7:     end if
 8:     L ←rightmost child of L
 9:     d ← d + 1
10: end while
11: /* State: L is rightmost leaf. P is lowest right border node with less than k children.
    d stores tree depth. d_P stores depth of P. */
12: if L is not max-data leaf then
13:     Add byte to L                                                    ▷ Case 1
14: else if P ≠ ⊥ then
15:     /* Add a leaf. */                                                ▷ Case 2
16:     C ←create chain of d − d_P − 1 inner nodes ending with a one-byte leaf
17:     Add C as a child to P
18: else
19:     /* It is a max-data tree. Increase tree depth. */                ▷ Case 3
20:     R′ ←create copy of node R
21:     C ←create chain of d inner nodes ending with a one-byte leaf
22:     Overwrite R with a new inner node having children {R′, C}
23: end if
```

Algorithm 1 Grow a balanced left-max-data tree by one byte. In lines 1-11, the algorithm finds the rightmost leaf and the lowest right border node P with less than k children. This is the node where the algorithm could add new nodes to grow the number of leaves. Lines 12-13 handle case 1, in which the rightmost leaf is not full yet. The algorithm grows it by one byte and is done. Lines 14-17 handle case 2, in which there was an right border node that is not full yet. It adds a new leaf to it, and, because all leaves have to be on the same level, a chain of inner nodes. Lines 18-22 handle case 3, in which there was no such right border node. It is a max-data tree. The algorithm grows the tree by placing a new root node on top of the old one. Because a blob is referenced by storing the block ID of the root node, these lines replace the old root node with the new one to keep this ID intact.

tree. In that case, the tree can either hold another leaf without increasing tree depth (case 2) or it is a max-data tree (case 3). The last case adds an additional root node on top of the old one. Because trees are referenced by their root nodes and references should not become invalid, the algorithm actually does not add a root node, but stores the new root node in the existing root block R while copying the old content of R into a new node R'.

Lemma 2.35
If T is a balanced left-max-data tree before calling Algorithm 1, then it is a balanced left-max-data tree afterwards.

Proof We describe the three cases separately.

Case 1: In this case, the rightmost leaf is not a max-data leaf. The algorithm added one byte to the last leaf without changing tree structure or the number of bytes in non-rightmost leaves. T still is a balanced left-max-data tree.

Case 2: Since P was the lowest inner node with less than k children before, all old children of P induce max-data trees. The new child induces a path where each node is a rightmost child of its parent. Since P is at depth d_P and the algorithm adds a chain consisting of $d - d_P - 1$ inner nodes and one leaf to P, the new leaf is at depth d and therefore at the same level as the other leaves. Since the new chain is added into the leftmost empty child slot in P, all added children are as far left as possible. The old rightmost leaf was a max-data leaf, so the only non-max-data leaf is the new rightmost leaf. All together, this fulfills Definitions 2.10 and 2.21. P still induces a left-max-data tree.

Since P is a right-border node, the path from it to the root consists only of right-border nodes. Inductively going up this path from P' to its parent P'', the situation is always that the P' (the rightmost child of P'') induces a left-max-data tree and the other children induce max-data-trees. So P'' also induces a left-max-data tree. Continuing this induction chain to the root node implies that T is a left-max-data tree. Since T was balanced before and all old children of the root are still present, T is still balanced; see Definition 2.25.

Case 3: Since there is no non-leaf right-border node with less than k children, T was a max-data tree. The new tree consists of a root node with two children. The first child induces T, which is a max-data tree. The second child induces a chain of d inner nodes and a leaf. This leaf is on the same level as the other leaves. Furthermore, each node on this chain is a rightmost child of its respective parent. All added children are as far left as possible. The old rightmost leaf was a max-data leaf, so the only non-max-data leaf is the new rightmost leaf. Overall, this fulfills Definitions 2.10 and 2.21. T still is a left-max-data tree. Since the new root has two children, T is still balanced; see Definition 2.25. ∎

2.10.3 Shrinking

The algorithm for shrinking a balanced left-max-data tree by one byte is very similar to the algorithm for growing it; see Algorithm 2. In line 12, the most important part to understand the idea of the algorithm is the check whether L is not a min-data leaf. Checking $L = R$ is for handling the corner case where a one-byte tree is shrunk to zero bytes. A one-byte leaf is a min-data leaf and usually the algorithm would not shrink it to zero bytes but instead remove the whole leaf, which is not what the algorithm should do in this corner case.

The following Lemma 2.36 shows the correctness of the algorithm.

Require: T stores at least one byte
1: $R \leftarrow$root node of T
2: $L \leftarrow R$, $P \leftarrow \perp$, $d \leftarrow 0$, $d_P \leftarrow 0$
3: **while** L has children **do**
4: **if** L has more than one child **then**
5: $P \leftarrow L$
6: $d_P \leftarrow d$
7: **end if**
8: $L \leftarrow$rightmost child of L
9: $d \leftarrow d + 1$
10: **end while**
11: /* State: L is rightmost leaf. P is lowest right border node with more than one child. d stores tree depth. d_P stores depth of P. */
12: **if** $L = R \vee L$ is not min-data leaf **then**
13: Remove byte from L ▷ Case 1
14: **else**
15: /* Remove a leaf. */ ▷ Case 2
16: Remove subtree induced by rightmost child of P
17: **if** R has only one child **then**
18: /* Decrease tree depth. */
19: Overwrite R with its child
20: **end if**
21: **end if**

Algorithm 2 Shrink a balanced left-max-data tree by one byte. Lines 1-11 find the rightmost leaf and the lowest right border node with more than one child. This is the node where the algorithm could remove the rightmost child to shrink the tree by exactly one leaf. Lines 12-13 handle case 1, in which the rightmost leaf stores more than one byte. The algorithm shrinks this leaf by one byte and is done. Lines 14-20 handle case 2, in which a leaf has to be removed to shrink the tree. Lines 17-20 take care that the tree stays balanced. If after the operation the root node has only one child left, it decreases the tree depth by replacing the root node with its child.

Lemma 2.36
If T stores at least one byte and is a balanced left-max-data tree before calling Algorithm 2, then it is a balanced left-max-data tree afterwards.

Proof We describe the two cases separately.

Case 1: If $L = R$, then the tree consists of exactly one node, and this node is a leaf. Since the precondition is that the tree stores at least one byte, this leaf stores at least one byte. It can be shrunk by one byte. If L is not a min-data leaf, the same is true. In both cases, the tree structure was not changed and no non-rightmost leaves have been changed. T is still a balanced left-max-data tree.

Case 2: This case is only executed if $L \neq R$, which means the root node has children, and L is a min-data leaf. To shrink the tree by one byte, the algorithm removes that leaf and all unnecessary inner nodes. Since T is balanced, the root node has at least two children.

That is, there is a $P \neq \perp$ which is the lowest right-border node with more than one child. After removing its rightmost child, P still induces a left-max-data tree.

Since P is a right-border node, the path from it to the root consists only of right-border nodes. Inductively going up this path from P' to its parent P'', the situation is always that the P' (the rightmost child of P'') induces a left-max-data tree and the other children induce max-data-trees. Thus, P'' also induces a left-max-data tree. Continuing this induction chain to the root node implies that T is a left-max-data tree.

It remains to show that T is balanced. Assume T is not balanced anymore after removing the rightmost child of P, that is, R has only one child. Since T was balanced before, the old root had more than one child and the leftmost child induced a perfect k-ary tree T'. Now the root has only one child left which leads to T'. In this case, lines 17-20 replace T with T', so in the end T is a perfect k-ary tree, which is balanced according to Lemma 2.27. ■

3. Cryptography Basics

This chapter introduces some cryptography basics. We introduce block ciphers, some modes of operation and some security definitions. Many definitions in this chapter are taken from the book of Katz and Lindell [KL08].

Ciphers that are secure on an information-theoretic level are difficult to use in practice. For ciphers that are not secure on an information-theoretic level, the ciphertext still contains information about the plaintext which allows an attacker to verify keys. So they always have a small probability of guessing the correct key. However, this probability can be so small that it is negligible. To define, which probabilities are acceptable, we define the concept of negligible functions. A negligible function is a function that shrinks faster than any polynomial. Using this, the attacker is allowed to have a success probability of $\mathsf{negl}(k)$ where negl is a negligible function and k is the security parameter; i.e. key size.

Definition 3.1 (Negligible Function)
A function negl *is negligible if for every polynomial* p *there exists an* N *such that for all integers* $n > N$ *it holds that* $|\mathsf{negl}(n)| < \frac{1}{\mathsf{p}(n)}$.

3.1 Security Definitions

For security proofs, it has to be defined what a secure system is and what an attacker is capable of. For this, a security goal and an attacker model are defined. The security goal can for example be to keep a specific message confidential. Or even stronger, it can be to keep the attacker from getting a single bit of information about the message. The goal can also be to ensure integrity of a message. The other point to consider is the attacker model. If there are no time constraints for attackers, then an attacker testing all possible keys is successful against every cryptographic scheme that is not secure on an information-theoretic level. Known solutions for security on an information-theoretic level bring a lot of disadvantages as well, so this is usually not wanted. Neither is it needed, because real world attackers are not able to use that much time.

3.1.1 Security Goals

Security goals are often formulated as games. In this section, we describe the two most common goals for encryption, namely confidentiality and integrity.

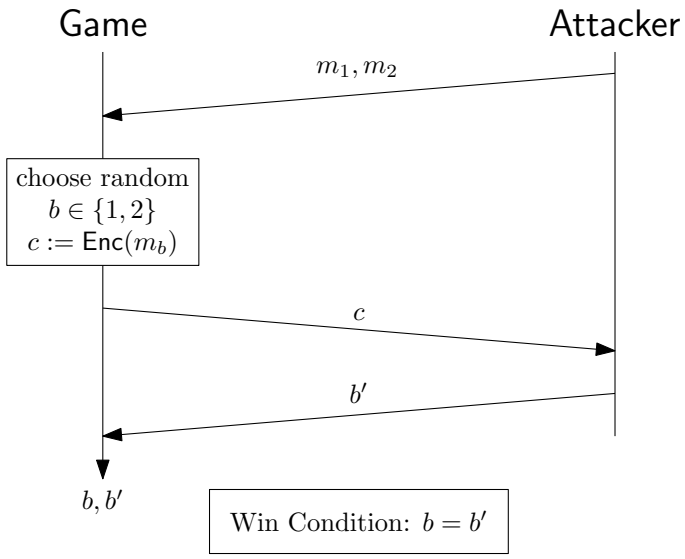


Figure 3.1 Indistinguishability game. An Attacker chooses two plaintexts m_1, m_2. One of these is encrypted and the ciphertext sent to the attacker. The attacker wins if it can tell which plaintext was encrypted. Oracle queries are not shown.

In the following, we define two confidentiality goals, namely *indistinguishability* and *non-malleability*. Later in Section 3.1.3, we define security goals for integrity.

Indistinguishability

When the security goal is to keep confidentiality, commonly the game of indistinguishability is used. An attacker can provide two plaintext messages m_1, m_2 and the game encrypts one of them at random and gives it to the attacker. They should then be unable to tell which of the messages was encrypted. In practice, this means that an attacker cannot get a single bit of information about the plaintext. The game is illustrated in Figure 3.1. In the end, the attacker chooses b', their guess of which message was encrypted. The attacker wins the game if $b = b'$. A blindly guessing attacker wins the game in half the cases, so we define the advantage of an attacker A as

$$\mathsf{Adv}_{\mathsf{IND}}(\mathsf{A}) := \mathsf{Pr}[b = b'] - \frac{1}{2}$$

The system is called *indistinguishable*, if the advantage of the attacker is negligible according to Definition 3.1.

Non-Malleability

For a stronger security goal, the game of non-malleability can be used. In Section 3.1.4, we show that non-malleability implies indistinguishability. There exist various equivalent definitions for non-malleability. We describe the definition of Bellare et al. [Bel+98].

For non-malleability, an attacker should not be able to modify a ciphertext $c = \mathsf{Enc}(m)$ in a way that the modified plaintext m' has some meaningful relation to the original plaintext m. So non-malleability can also be used as a weak security definition for integrity. With

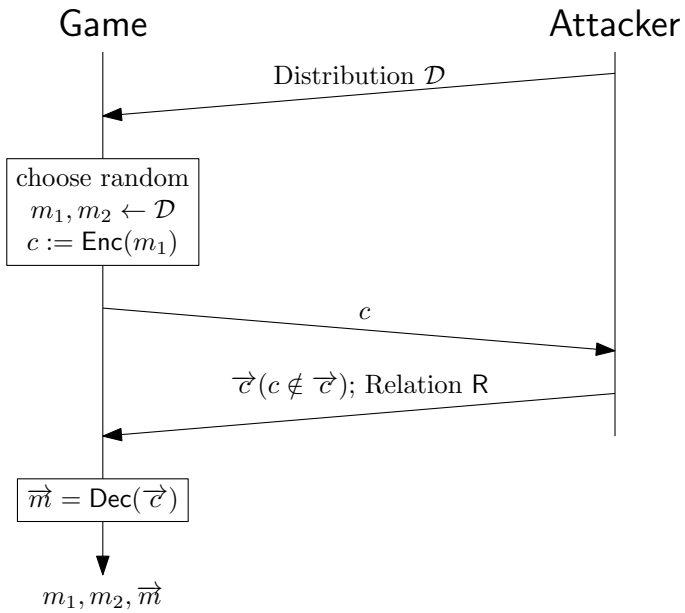


Figure 3.2 Non-malleability game. An attacker determines a plaintext distribution of which the game draws two plaintexts, m_1, m_2. m_1 is encrypted and the ciphertext sent to the attacker. The attacker wins if it can give a vector of new ciphertexts $\overrightarrow{c}$ and a relation R that relates their plaintexts to m_1 but not to m_2. Oracle queries are not shown.

non-malleability, ciphertext manipulations are not necessarily recognized, but it is ensured that the resulting plaintext does not make any sense.

Figure 3.2 illustrates the game that works as follows: The attacker chooses a distribution $\mathcal{D}$ over the possible plaintexts and gives it to the game. The distribution must be valid, which means it gives non-zero probability only to plaintexts of the same length. The game draws two plaintexts m_1, m_2 according to this distribution, encrypts m_1 and sends the ciphertext c to the attacker. Then, the attacker chooses a vector $\overrightarrow{c}$ of ciphertexts and sends it back to the game. The challenge c is not allowed to be contained in $\overrightarrow{c}$. Also, all ciphertexts in $\overrightarrow{c}$ must be decryptable. Let $n := |\overrightarrow{c}|$ be the number of entries in $\overrightarrow{c}$. The attacker also chooses and sends the description of an arbitrary $n+1$-ary relation R defined on plaintexts. The game decrypts the entries of $\overrightarrow{c}$ individually to $\overrightarrow{m}$ and checks whether $\mathsf{R}(m_1, \overrightarrow{m})$ or $\mathsf{R}(m_2, \overrightarrow{m})$. The attacker wins the game, if the probability for $\mathsf{R}(m_1, \overrightarrow{m})$ is significantly different than the probability for $\mathsf{R}(m_2, \overrightarrow{m})$. If they win, it means that the attacker could forge encrypted plaintexts $\overrightarrow{m}$ while knowing their relation to the challenge plaintext m_1. The advantage of an attacker A is defined as

$$\mathsf{Adv}_{\mathsf{NM}}(\mathsf{A}) := |\mathsf{Pr}[\mathsf{R}(m_1, \overrightarrow{m})] - \mathsf{Pr}[\mathsf{R}(m_2, \overrightarrow{m})]|$$

The system is called *non-malleable*, if the advantage of the attacker is negligible according to Definition 3.1.

3.1.2 Attacker Models

An attacker can have different capabilities. All attackers have in common, that they have to be probabilistic polynomial time algorithms. That is, they can use polynomial time

and a random generator. The random generator is important, because for a scheme to be insecure, attackers do not have to succeed every time. It is already insecure if an attacker succeeds with non-negligible probability. For the attacker, this is easier to achieve if they can use randomized algorithms.

Chosen Plaintext Attack (CPA)

In the CPA *(chosen plaintext attack)* model, the attacker gets an encryption oracle. Before sending plaintexts m_1, m_2 to the distinguishability game (or the distribution $\mathcal{D}$ to the non-malleability game), they can send an arbitrary number of requests to the encryption oracle which then encrypts them and returns the corresponding ciphertexts. They are even allowed to encrypt m_1 and m_2. Because of this, any deterministic algorithm cannot be safe against such an attacker. The attacker is also allowed to do some calculations between oracle requests, they do not have to send all oracle requests at once. The number of requests is implicitly polynomially bounded because of the time restriction on the attacker.

Giving an attacker an encryption oracle seems to be too powerful and unnecessary at first, but it is a security advantage if the system can be proven secure against a stronger attacker. Furthermore, an attacker in practice actually often has an encryption oracle. In many security protocols, an attacker can make another party encrypt certain plaintexts (for example a greeting message with the attackers name in it) and intercept the results.

This attacker model can be combined with the security goals to IND-CPA or NM-CPA to get indistinguishability or non-malleability against attackers with an encryption oracle.

Nonadaptive Chosen Ciphertext Attack (CCA1)

In the CCA1 *(nonadaptive chosen ciphertext attack)* model, the attacker is stronger than in the CPA model. They get an encryption and a decryption oracle. Before sending plaintexts m_1, m_2, or the distribution $\mathcal{D}$, they can send an arbitrary number of requests to encryption and decryption oracles, encrypting and decrypting the given plaintexts or ciphertexts as wished. Any system that is secure against a CCA1 attacker is also secure against a CPA attacker, because a CCA1 attacker has any capabilities a CPA attacker has.

For many symmetric encryption schemes, encryption and decryption are identical algorithms. In that case, encryption and decryption oracles are identical and security against IND-CCA1 (NM-CCA1) is equivalent to security against IND-CPA (NM-CPA).

Adaptive Chosen Ciphertext Attack (CCA2)

If the attacker should be even stronger, the CCA2 *(adaptive chosen ciphertext attack)* model can be used. The attacker can use their encryption and decryption oracles at any time, even after they received c from the game. They can however not query the decryption oracle for c, otherwise the game becomes trivial. This is the strongest attacker model defined here. Any system that is secure against a CCA2 attacker is also secure against a CCA1 or CPA attacker, because a CCA2 attacker has any capabilities the other attackers have.

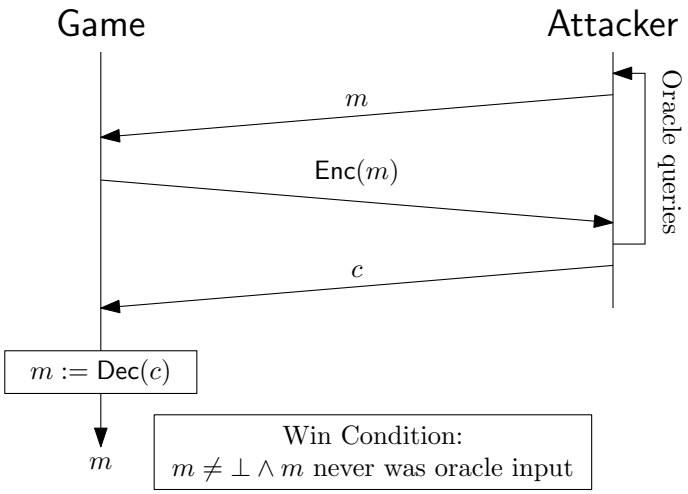


Figure 3.3 Game for INT-PTXT security. After using an encryption oracle, an attacker has to fake a valid ciphertext c that decrypts to a plaintext that never was oracle input.

3.1.3 Integrity

Non-malleability ensures that an attacker cannot forge a meaningful plaintext. However, attackers can still forge non-meaningful plaintexts. The goals of the security definitions in this chapter is to entirely prevent an attacker from forging messages. This can for example be implemented using Message Authentication Codes [Bla00]. The security definitions in this section only provide integrity, not confidentiality. A scheme sending the unencrypted plaintext together with a good Message Authentication Code can achieve the integrity definitions, but not any of the confidentiality definitions we introduced.

Integrity of Plaintexts (INT-PTXT)

The goal of INT-PTXT (Integrity of Plaintexts) is to prevent forged plaintexts. That is, every ciphertext that validly decrypts has to decrypt to a plaintext that was previously encrypted knowing the secret key.

The game is illustrated in Figure 3.3. In the INT-PTXT game, an attacker gets an encryption oracle and can encrypt an arbitrary number of plaintexts. The attacker is also allowed to do some calculations between oracle requests, it does not have to send all oracle requests at once. Once it is ready, it sends a ciphertext c to the game. The game is won, if the ciphertext decrypts successfully to a plaintext that was not previously sent to the encryption oracle. With M being the set of plaintexts encrypted by the oracle, the advantage of an attacker A is defined as

$$\mathsf{Adv}_{\mathsf{INT\text{-}PTXT}}(\mathsf{A}) := \mathsf{Pr}[\mathsf{Dec}(c) \neq \perp \wedge \mathsf{Dec}(c) \notin M]$$

Integrity of Ciphertexts (INT-CTXT)

The INT-PTXT definition ensures that if a decryption is successful, it decrypts to a plaintext that was previously encrypted knowing the secret key. It is still possible for an attacker to

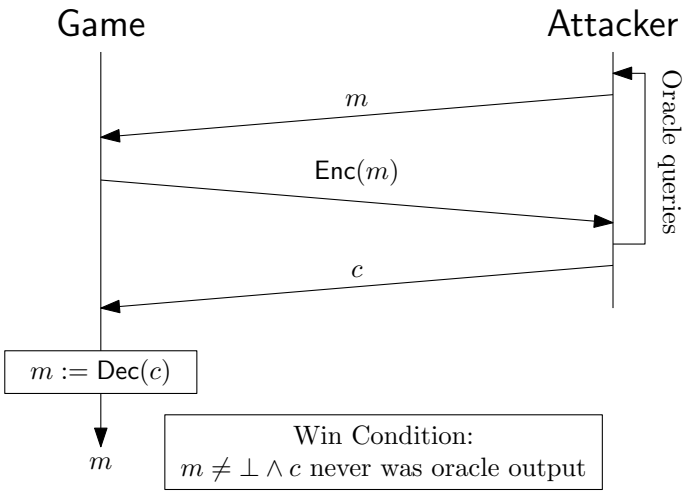


Figure 3.4 Game for INT-CTXT security. After using an encryption oracle, an attacker has to fake a valid ciphertext c that never was oracle output.

forge a new ciphertext, as long as it decrypts to a previously encrypted plaintext. To also prevent that, INT-CTXT goes a step further and prevents any kind of forgery.

The game works very similar to the INT-PTXT game and is illustrated in Figure 3.4. The only difference is the win condition. The INT-CTXT game is won, if the ciphertext decrypts successfully and is different to all ciphertexts output by the encryption oracle. With C being the set of ciphertexts output by the oracle, the advantage of an attacker A is defined as

$$\mathsf{Adv}_{\mathsf{INT\text{-}CTXT}}(\mathsf{A}) := \mathsf{Pr}[\mathsf{Dec}(c) \neq \perp \wedge c \notin C]$$

3.1.4 Relations

This section introduces some relations between the security definitions introduced.

atk-CCA2 $\Rightarrow$ atk-CCA1 $\Rightarrow$ atk-CPA

As stated before, a cryptographic scheme that is secure against a CCA1 attacker is also secure against a CPA attacker, because the CCA1 attacker can do everything a CPA attacker can do. The same argument holds for CCA2 and CCA1.

NM-atk $\Rightarrow$ IND-atk

A more interesting result is that security against non-malleability implies security against indistinguishability. An attacker who can recognize plaintexts can also use the encryption oracle to generate a plaintext that has a certain relation to the recognized plaintext [Bel+98].

IND-CCA1 $\not\Rightarrow$ NM-CPA

There is no relation between IND-CCA1 and NM-CPA. There are cryptographic schemes fulfilling either of them without fulfilling the other [Bel+98].

IND-CCA2 ⇒ NM-CCA2

IND-CPA does not imply NM-CPA and neither does IND-CCA1 imply NM-CCA1. However, an attacker who wins the NM-CCA2 game, can modify ciphertexts while knowing a relation between them. Using this, they can win the IND-CCA2 game by transforming the challenge c, letting the decryption oracle decrypt it, and checking the relation. That is, IND-CCA2 implies NM-CCA2 and therefore also IND-CCA1, NM-CCA1, IND-CPA and NM-CPA [Bel+98]. So IND-CCA2 (or NM-CCA2) is the strongest confidentiality definition among those we defined. If a system is IND-CCA2 secure or NM-CCA2 secure, it also fulfills all other confidentiality definitions.

INT-CTXT ⇒ INT-PTXT

If an attacker cannot forge any valid ciphertexts, it also cannot forge ciphertexts decrypting to new plaintexts. That is, INT-CTXT implies INT-PTXT [BN08].

INT-CTXT ∧ IND-CPA ⇒ IND-CCA2

Given a scheme that is IND-CPA and INT-CTXT secure, then it is also IND-CCA2 secure [BN08]. That is, to get an IND-CCA2 secure scheme, it is enough to take an IND-CPA secure scheme and add authentication to make it INT-CTXT secure. This is especially useful, since IND-CCA2 is the strongest of our confidentiality definitions.

3.2 Block Ciphers

A block cipher encrypts and decrypts blocks of data. In our definition, it encrypts plaintexts with n bytes to ciphertexts with n bytes using k bytes as key. This section is based on the definitions in the book of Katz and Lindell [KL08] with minor changes.

For an ideal cipher, without the key, a ciphertext should not be distinguishable from a random sequence of bytes of the same length. Since this should be true for any encrypted plaintext, encryption with a fixed key should ideally be a random function. Because block ciphers encrypt fixed-size blocks into blocks of the same size, it is an endofunction. And since encryption should be reversible, it is a bijective endofunction. That is, encryption should ideally be a random permutation. Since random permutations are hard to construct, we define pseudorandom permutations, a class of permutations that can be distinguished from random permutations only with negligible probability.

Definition 3.2 (Pseudorandom Permutation)
Let A be any set and $\mathsf{f} : A \to A$ be a random permutation. A pseudorandom permutation is $\mathsf{F} : A \to A$ with the following conditions: F and F^{-1} are computable in polynomial time and for any probabilistic polynomial-time distinguisher D, there exists a negligible function negl such that:

$$|\mathsf{Pr}[\mathsf{D}(\mathsf{f}) = 1] - \mathsf{Pr}[\mathsf{D}(\mathsf{F}) = 1]| \leq \mathsf{negl}(n)$$

For any fixed key, the block cipher should be a pseudorandom permutation. This leads to the following definition of block ciphers.

Definition 3.3 (Block Cipher)
Let $n \in \mathbb{N}$ be the block size and $m \in \mathbb{N}$ the key length. A block cipher is $\mathsf{F} : \{0,1\}^m \to (\{0,1\}^n \to \{0,1\}^n)$ where for any $k \in \{0,1\}^m$, $\mathsf{F}(k)$ is a pseudorandom permutation. To simplify notation, we define $\mathsf{F}_k := \mathsf{F}(k)$.

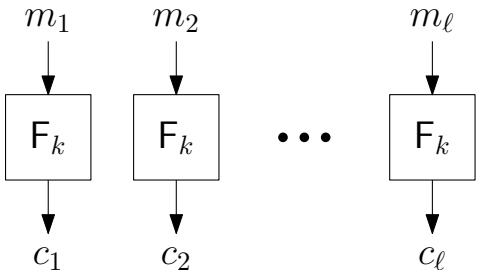


Figure 3.5 Block cipher F_k in ECB mode. Each block is encrypted individually.

A plaintext block $p \in \{0,1\}^n$ can be encrypted by computing $\mathsf{F}_k(p)$ and a ciphertext block $c \in \{0,1\}^n$ can be decrypted by computing $\mathsf{F}_k^{-1}(c)$.

In cryptographic theory, it is not proven that such block ciphers do exist. For common block ciphers like AES, the prevailing assumption is that they fulfill this definition, but this is nonetheless an assumption. Security proofs in this thesis also rely on this assumption.

3.3 Modes of Operation

A block cipher itself only operates on fixed size blocks and is deterministic. It being deterministic significantly reduces security and it operating on fixed size blocks yields bad usability. Both problems can be solved by using block ciphers in certain modes of operation as presented in this section. All modes presented have the task of encrypting a plaintext consisting of an arbitrary number ℓ of n-byte plaintext blocks $m_1, \ldots, m_\ell$ into ciphertext blocks $c_1, \ldots, c_\ell$.

3.3.1 Electronic Codebook Mode (ECB)

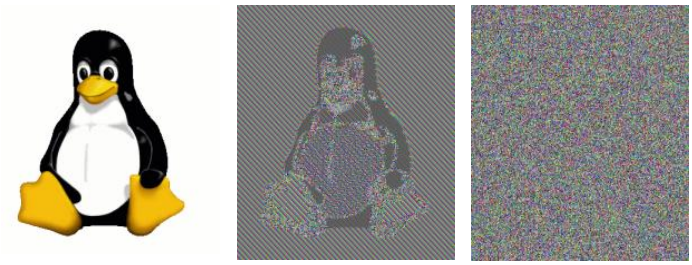
The simplest—but insecure—idea to encrypt arbitrarily sized data is to encrypt each m_i individually; see Figure 3.5.

$$\forall i \in \{1, \ldots, \ell\} : c_i := \mathsf{F}_k(m_i)$$

This mode is called Electronic Codebook Mode, because each block can independently be looked up from a codebook containing the block cipher evaluations.

The main security issue with this approach is that encryption of a block is deterministic and depends only on the key and on the block itself. That is, if the same plaintext block is encrypted with the same key, it results in the same ciphertext block. If a chain of plaintext blocks is encrypted where some plaintext blocks are equal, the corresponding ciphertext blocks are also equal. This way, an attacker can easily recognize whether the plaintext has repetitions. Furthermore, plaintext structure stays intact; see for example the picture in Figure 3.6, which was encrypted in ECB mode and CBC mode for comparison. The intact structure also allows statistical attacks like an analysis of the frequency of certain blocks.

Because ECB mode is deterministic, it cannot be IND-CPA secure. An attacker can always choose m_1, m_2, encrypt them with an encryption oracle to get their deterministic ciphertext c_1, c_2, and use them to later recognize which of the plaintexts was encrypted. Because it is not IND-CPA secure and all of IND-CCA1, NM-CCA1, IND-CCA2, NM-CCA2 imply IND-CPA security, ECB mode also does not fulfill any of these security definitions.



Original image ECB mode CBC mode

Figure 3.6 An image encrypted in ECB and CBC mode. ECB mode does not hide the plaintext structure. Source: [Ewi]

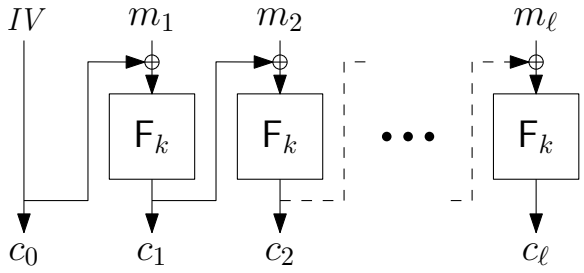


Figure 3.7 Block cipher F_k in CBC mode. The ciphertext of a block is xored to the plaintext of the next block before encryption.

3.3.2 Cipher Block Chaining Mode (CBC)

The Cipher Block Chaining Mode is more secure, because it is indeterministic and avoids statistical attacks by chaining blocks together. A (pseudo)random initialization vector (IV) is xored to the first plaintext block before it is encrypted. For later blocks, the mode xores the ciphertext of the previous block instead of the initialization vector; see Figure 3.7. The initialization vector itself is needed for decryption, so it has to be part of the ciphertext output. This allows the following compact definition.

$$c_0 := IV$$

$$\forall i \in \{1, \ldots, \ell\} : c_i := \mathsf{F}_k(m_i \oplus c_{i-1})$$

Ciphertexts can be decrypted by reversing the process as in the following formula.

$$\forall i \in \{1, \ldots, \ell\} : m_i := \mathsf{F}_k^{-1}(c_i) \oplus c_{i-1}$$

It is crucial that the initialization vector must be unpredictable at encryption-time and not just unique (see the TLS CBC IV attack [Moe04]). This was not the case in earlier SSL/TLS implementations and allowed the BEAST attack [DR11], a severe security vulnerability. When the block cipher is a pseudorandom permutation as defined above and CBC mode is used with a pseudorandom initialization vector, then it is IND-CPA secure, but not IND-CCA2 and it does not fulfill any non-malleability security definition [Bel+97].

A drawback of CBC mode is that encryption is inherently sequential. To encrypt a block, all predecessor blocks have to be encrypted. Decryption can be parallelized and it is also

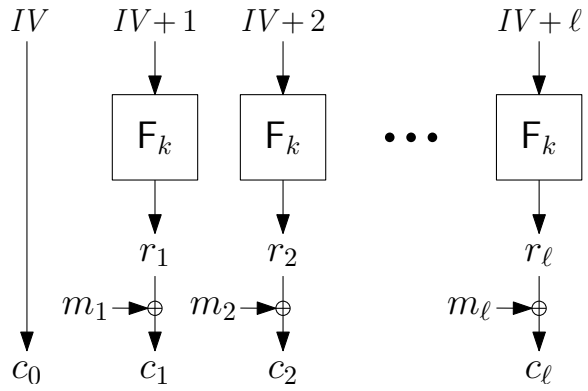


Figure 3.8 Block cipher F_k in CTR mode. Encrypting a running counter generates a pseudorandom stream which is xored with the plaintext.

possible to decrypt one block from the middle without having to decrypt other blocks. A bit error in a ciphertext block destroys the block containing it and because of the feedback loop also the next block.

3.3.3 Counter Mode (CTR)

In this mode, the block cipher works independently from the plaintext and generates a pseudorandom stream, which is then xored to the plaintext; see Figure 3.8. A (pseudo)random initialization vector (IV) is chosen and a pseudorandom stream $r_i := \mathsf{F}_k(IV + i)$ generated by applying the block cipher F_k to increments of that counter. A counter increment is an integer addition modulo 2^n. Then, the ciphertext is produced as an XOR between the plaintext and the pseudorandom stream.

$$c_0 := IV$$

$$\forall i \in \{1, \ldots, \ell\} : c_i := m_i \oplus r_i = m_i \oplus \mathsf{F}_k(IV + i)$$

In CTR mode, as opposed to CBC mode, encryption and decryption are both parallelizable, because there are no dependencies between the blocks. Furthermore, if a bit error is happening outside of the IV region, then it only affects this one bit and does not affect any other bits.

Like CBC mode, given the block cipher is a pseudorandom permutation, CTR mode is IND-CPA secure, but not IND-CCA2 and it does not fulfill any non-malleability security definition [KL08].

3.3.4 Galois Counter Mode (GCM)

As described before, flipping a single bit in the ciphertext in CTR mode just flips the same bit in the plaintext without affecting other bits. This is great for error containment, but is not desirable for integrity purposes. An attacker could easily flip arbitrary plaintext bits while the other bits stay intact. In CBC mode, when an attacker modifies a ciphertext bit, this usually scrambles the content of the corresponding and the next plaintext block. This is better, but not optimal yet.

Galois Counter Mode [Dwo07] is a mode of operation that not only ensures confidentiality, but also integrity. It works exactly like CTR mode, but additionally computes an auth tag that is stored with the ciphertext. On decryption, the auth tag is recomputed and checked for validity. The computation of the auth tag uses multiplications in the galois field. Details can be found in Morris Dworkin et al [Dwo07] or David A. McGrew et al [MV04].

By generating the auth tag during encryption, GCM has some advantages over methods that handle authentication separately from encryption. Firstly, it is easier to get the implementation right, because it is already built in and the implementer does not have to think about decisions like whether the auth tag is built for the plaintext or for the ciphertext. Secondly, it has better performance since GCM can compute the auth tag with only little overhead additional to the computations it has to do for encryption anyhow.

Since encryption is done with CTR mode, GCM mode is IND-CCA1 secure given that the block cipher is a pseudorandom permutation. GCM is not proven to be INT-CTXT secure, but has been proven to be secure in the concrete security model [MV04]. If GCM mode is assumed to be INT-CTXT secure, it is also IND-CCA2 and NM-CCA2 secure, see Section 3.1.4.

There are other modes like CCM mode, which combines CTR mode with a CBC-MAC and is proven to be INT-CTXT and IND-CCA2 secure [Fou+08; BN08], but they have worse performance than GCM mode, because they calculate the MAC independently from the ciphertext.

3.4 Conclusion

The goal for CryFS is IND-CCA2 and INT-CTXT security. These are the strongest security definitions we defined for confidentiality and integrity.

In Section 7, we show that that CryFS fulfills IND-atk security if the underlying block cipher fulfills IND-atk security ($\mathsf{atk} \in \{\mathsf{CPA}, \mathsf{CCA1}, \mathsf{CCA2}\}$). Under an additional assumption, we show that the same is true for INT-atk security ($\mathsf{atk} \in \{\mathsf{CTXT}, \mathsf{PTXT}\}$).

4. System Design

The goal of this thesis is to design an encrypted filesystem that can be used to synchronize files and folders between multiple clients securely over the cloud. Each client allows access to plaintext files, while in the background the clients synchronize with each other, transmitting only ciphertexts. Our system focuses on the encryption layer and stores ciphertext files in a folder that can be processed by third party synchronization software like Dropbox, Rsync, Unison or others. The user workflow looks like they are using the third party synchronization client directly without encryption. But actually, the synchronization service does not have to be trusted and is prevented from accessing the plaintext files. Although the main goal is to work together with third party synchronization tools like Dropbox that keep a local copy of the synchronized files, it is also possible to modify the system so that it does not keep a local copy of the ciphertexts but stores the blocks remotely, for example on NFS or Amazon S3.

It is assumed that the synchronization client is trusted and the attacker is on server side, because it is difficult to keep a locally installed synchronization client from accessing the plaintext data when other locally installed applications (like a text editor) should be able to access them. There are actually solutions to achieve that: the synchronization client could for example be run under a system user that does not have access to the plaintext data. Another possible solution is application isolation as announced for future Ubuntu 16.10. However, this is out of the scope of this thesis. Under the assumption of a trusted client, the system protects against an attacker that controls the synchronization server.

4.1 General Idea

To keep an attacker from getting information about file sizes, everything stored is broken down into blocks of the same size. Each block is stored in one encrypted real file and synchronized by the cloud synchronization client. Our implementation has a configurable block size. According to our experiments, a block size of 8-32KB (depending on the system) yields the best performance.

Large files are split into multiple blocks. Filesystem metadata and directory structure is also stored using these blocks.

All blocks being the same size is one foundation of the security of the filesystem. An attacker is not able to see file or directory sizes without being able to decrypt blocks.

4.2 Design Goals

The following is a list of goals we had in mind when designing CryFS. While they describe an optimal solution, we also explain to which degree these goals are achieved.

Transparency	In their daily workflows, users should not notice any difference to using third party synchronization directly. They should be able to work on plaintext files as if they were only plaintext files.
Local Performance	Adding encryption always has a performance impact. However, the impact should be reasonable to keep the system useful.
Synchronization Compatibility	The system should work correctly together with third party synchronization clients, e.g. avoid synchronization conflicts.
Backend Flexibility	The system should be flexible about the way synchronization works. It should allow storing the encrypted data locally (for third party synchronization) or directly on remote servers (e.g. NFS, S3).
Platform Independence	It should run on any device or operating system.
Network Performance	The amount of needed network transmission should be small when only little data has changed.
Storage Efficiency	The amount of space needed (ciphertext size) should not be much higher than the amount of space used (plaintext size).
Content Confidentiality	An attacker should not get any information about the file contents.
Content Integrity	An attacker should not be able to manipulate the stored files without the user noticing it.
Metadata Confidentiality	An attacker should not get any information about filesystem metadata (e.g. file attributes).
Metadata Integrity	An attacker should not be able to manipulate filesystem metadata without the user noticing it.
Structural Confidentiality	An attacker should not get any information about the size of individual files or about directory structure.
Structural Integrity	An attacker should not be able to manipulate file sizes or directory structure without the user noticing it.
Partial Shareability	The user should be able to share a subset of files/folders with a friend without giving them access to other files.

4.2.1 Achieving the Goals

Transparency

There are multiple ways to achieve transparency. The simplest solution is to implement a virtual filesystem. There is a real folder containing real files on the hard disk, and based on them, a virtual folder with virtual files. The ciphertext data could for example be stored in the real folder and offer a plaintext view in the virtual folder.

When the user accesses the virtual folder, they get their folders and directories en- and decrypted transparently by the virtual filesystem driver. Many other systems like EncFS, eCryptFS, TrueCrypt and VeraCrypt take the same approach.

Forward vs. Reverse-Encryption

An alternative approach is reverse encryption, where the plaintext is kept in real files on the hard disk and the virtual filesystem driver offers a virtual folder with encrypted files. The synchronization service is pointed to the virtual folder and therefore still synchronize only the ciphertexts. The advantage of reverse encryption is that accessing plaintext files is faster because it is a direct hard disk access and there is no encryption envolved.

But it also has some disadvantages. In case the virtual filesystem is not mounted, the synchronization client might see an empty directory and—assuming the user deleted all files—delete them on the server. Furthermore, each access from the synchronization client to one of the encrypted files should yield the same ciphertext or otherwise the client keeps re-uploading everything. So the system has to keep metadata like initialization vectors somewhere. Additionally to initialization vectors, the system includes some more random decisions in the encryption step, for example random block IDs, which also have to be kept as metadata.

Forward encryption proved to be quite fast in our experiments and we think the added complexity of reverse encryption is not worth the performance gains. Using forward encryption, the system is also useable in a broader number of applications. It can for example also be used for local encryption if the goal is to protect data from someone stealing the hard disk.

Local Performance

A key point for local performance is using fast encryption algorithms and allowing parallel encryption when writing multiple files or one large file. This is achieved by encrypting on a per-block level. When writing to a big file, all touched blocks are independent and can be encrypted in parallel. This also means that if the user accesses or modifies a small part of a big file, only the few blocks containing this small part have to be decrypted or re-encrypted. By keeping a plaintext cache catching multiple changes to the same block, the system takes care that it does not encrypt more often than actually necessary. Encryption only happens when the block is thrown out of the cache.

Another aspect to consider is the way the filesystem is connected to the operating system. A simple way to implement such virtual filesystems is using user space filesystems like libFUSE[1] on Linux/Mac and Dokan[2] on Windows. Implementing the virtual filesystem

[1] http://fuse.sourceforge.net/
[2] http://dokan-dev.github.io/

in kernel space has a better performance, but it is harder to implement. CryFS uses the user space approach, but is implemented in a way that it could be easily ported to other paradigms.

Synchronization Compatibility

The classic use case is to use a third-party synchronization client like Dropbox that performs deferred synchronization. When designing a system that should work well with such clients, some things have to be kept in mind.

Hard disk encryption schemes like dm-crypt[3] that store the ciphertext on chunks of disk space do not offer access to the ciphertext as a file hierarchy. So third party synchronization clients—which are programmed to synchronize files—cannot use this data.

Some file backed encryption schemes (like eCryptFS[4]) assume they are the only ones accessing the real files on the hard disk. This way, they can gain a bit performance by using a lot of caching. However, third party synchronization clients also access the real files on the hard disk to synchronize them. If the lifetime of a cache entry is 5 minutes, then after changing a file, it takes 5 minutes for it to be actually written to the disk and to be synchronized to other clients. This might be a problem if the user wants to switch clients and immediately continue working on the same file. Even worse, if another clients modifies the file during these 5 minutes, only one version survives the conflict. This is why such systems do not work very well with third party synchronization clients. CryFS still uses caching for performance, but not as aggressively. By using very short cache timeouts (< 1 sec), it avoids such conflicts.

Conflicts generally have to be reduced to a minimum. Preventing conflicts where two clients are modifying the same file at the same time are inherently difficult and out of the scope of this thesis, but this should be the only type of conflict that can arise. Conflicts when two clients are modifying the same directory at the same time, for example both adding a new file to it, can be prevented.

As described before, the system splits large files into many blocks. At first glance, it seems good to extend this idea by combining multiple small files into a block to prevent wasting space. CryFS does not do that, because two clients modifying distinct files that are in the same block would modify the same ciphertext block and cause a conflict. Depending on the way the third party synchronization client handles conflicts, in the best case one of the versions survives.

Another important point is that the system has to be able to handle incomplete data. If no other entity accesses the real files, this cannot happen. But a cloud synchronization client downloads files from the cloud file by file and the system already sees some blocks before all blocks are there. So it is important that the system does not crash when there are only some of the blocks of a file or directory available.

Backend Flexibility

As said before, the classic use case is to store the ciphertext files locally and have them synchronized using a third-party cloud client. But the system is not restricted to this use

[3] https://gitlab.com/cryptsetup/cryptsetup/wikis/DMCrypt
[4] http://ecryptfs.org/

case. It is also possible to not store the encrypted data locally at all, but use network storage or direct cloud storage solutions (e.g. Amazon S3) for this. These storage solutions often do not have a filesystem interface, but offer to store and retrieve blocks of data using an API. This is enough for CryFS to work.

Platform Independence

Each operating system has different ways how filesystems have to be implemented. For CryFS to run on any platform, we offer a generic filesystem implementation using a platform independent interface, and a layer that maps this interface to platform dependent filesystem APIs. At the time of writing, this layer is implemented for Linux and Macintosh and we plan to implement it for Windows in future. We also plan to adapt the Linux layer for running it on Android.

Network Performance

Systems like TrueCrypt[5] or VeraCrypt[6] store the whole filesystem in one big file. So if the user changes one small virtual file, third party synchronization software most likely reuploads the big file with the whole filesystem. This has a bad performance and makes these filesystems not useable for cloud synchronization.

CryFS takes care that local plaintext changes cause local ciphertext changes. If a user changes one virtual file, the cloud synchronization client only has to reupload the ciphertext belonging to this file. If a user only changes a small part of a large virtual file, the cloud synchronization client only has to reupload the few blocks containing the related part of the ciphertext. The last point is also an advantage over EncFS, where a small change in a large virtual file leads to a re-encryption of the whole virtual file.

Storage Efficiency

As shown in Section 2.8, CryFS has little space overhead. Furthermore, the space needed is proportional to the space used and it grows dynamically. Filesystems like TrueCrypt, VeraCrypt or dm-crypt require allocation of a container with the maximal filesystem size in advance.

Content Confidentiality

CryFS is designed to keep attackers from getting information about the contents stored in files. This is a quite straightforward design goal and it also is fulfilled by most alternatives, e.g. EncFS[7] if the implementation weaknesses [Hor14] are ignored. However, to the best of our knowledge CryFS is the only one with a security proof; see Section 7.3.

Content Integrity

With this goal, attackers are kept from being able to modify the content of a file without clients noticing it. This is theoretically implemented by EncFS, but their implementation is flawed. The user can enable/disable the functionality with a flag and this flag is stored in a configuration file, which is by default stored together with the ciphertext. So an adversary can easily switch the flag to false without the user necessarily noticing it. CryFS provably fulfills content integrity; see Section 7.4.

[5] http://truecrypt.sourceforge.net/
[6] https://veracrypt.codeplex.com/
[7] http://www.arg0.net/#!encfs/c1awt

Metadata Confidentiality

This goal keeps attackers from being able to get information about file metadata like file attributes or permission bits. Sector level encrypted filesystems usually achieve this, since they encrypt the whole filesystem information, including metadata. EncFS on the other hand does not try to achieve this goal. They translate file attributes and any other metadata 1:1 to the encrypted files. It is their design choice to keep this data visible. CryFS provably fulfills metadata confidentiality; see Section 7.3.

Metadata Integrity

Similar to before, this goal keeps attackers from being able to modify file metadata without clients noticing it. CryFS also provably achieves this goal; see Section 7.4.

Structural Confidentiality

CryFS provably keeps attackers from being able to get information about file sizes or the directory structure; see Section 7.3. Like for *Metadata Confidentiality*, sector level encrypted filesystems achieve this by encrypting on a very low level. EncFS keeps the directory structure intact and only encrypts the file content. They also offer to encrypt the filename, but file size and directory structure stays unencrypted. As with *Metadata Confidentiality*, it is not their design choice to keep this hidden.

Using this information, an adversary can guess a lot about the files stored there. They can for example easily identify a given public set of files that belong together in a certain structure, e.g. the contents of an operating system installation medium.

Even if it is not a given public set of files, they can guess the kind of content that is stored there. A directory where all subdirectories have about 20 files and each file has a size of about 3 MB is very likely a music CD collection.

Structural Integrity

In CryFS, attackers are provably prevented from modifying the directory structure (moving files to different directories, changing filenames or sizes, undeleting files, ...) or rolling the whole filesystem back without clients noticing; see Section 7.4. EncFS offers to encrypt filenames depending on the whole file path, which is an approach to keep an attacker from moving files to different directories. They do not perform integrity checks on filenames and do not prevent undeleting files or rolling the whole filesystem back to a previous version.

Partial Shareability

This goal is about sharing a subset of files or directories with a friend, without having to share any other files. This is actually a goal that is hard to achieve. The reference implementation does not offer this, but we elaborate in Section 4.6.5 how this functionality could be added.

Comparison

Figure 4.1 shows an overview over the different filesystems and which goals they fulfill.

Legend

+ Fulfilled

– Not fulfilled

	CryFS	EncFS	eCryptFS	TrueCrypt, VeraCrypt	dm-crypt
Transparency	+	+	+	+	+
Local Performance	+	+	+	+	+
Synchronization Compatibility	+	+	–	–	–
Backend Flexibility	+	–	–	–	–
Platform Independence	+	+	–	+	–
Network Performance	+	+	+	–	n/a[2]
Storage Efficiency	+	+	+	–	n/a[2]
Content Confidentiality	+	+[1]	+	+	+
Content Integrity	+	–	–	–	–
Metadata Confidentiality	+	–	–	+	+
Metadata Integrity	+	–	–	–	–
Structural Confidentiality	+	–	–	+	+
Structural Integrity	+	–	–	–	–
Partial Shareability	–[3]	–	–	–	–

[1] Current implementation not secure in a cloud environment, but future versions are announced to be.
[2] dm-crypt does not allow access to the underlying ciphertexts for network synchronization.
[3] Partial Shareability is possible and planned for future versions.

Figure 4.1 Overview over the available filesystems and what goals they fulfill. Dm-crypt does not offer access to the underlying ciphertexts and can therefore not be used with cloud synchronization tools. TrueCrypt and VeraCrypt are also difficult to use in a cloud environment because of synchronization conflicts. eCryptFS causes undefined behaviour if synchronization software modifies the ciphertexts. EncFS does not support integrity and does not keep metadata or directory structure confidential.

4.3 Design Overview

Figure 4.2 shows the general system design. It follows a layered architecture, each layer only using the layer directly below and offering some functionality to the layer directly above. In the following, we describe the different layers.

Blockstore This layer is able to store/load data in blocks. Each block has to have the same size. An implementation can for example store blocks in local files on the hard disk (to be synchronized by third-party software), or store blocks directly using a network or cloud service (e.g. NFS or Amazon S3). This is the layer that achieves the goal of Backend Flexibility.

Blobstore This layer offers the functionality to store arbitrarily sized blobs ("binary large objects"), which can also be resized. It maps these blobs to the blocks of a blockstore. Blobs that are larger than a block are split into multiple blocks.

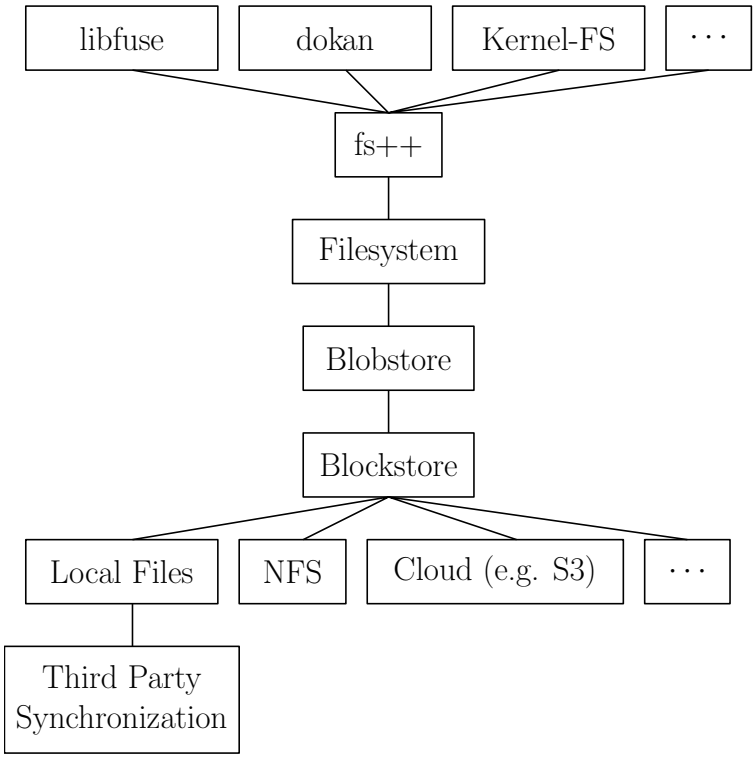


Figure 4.2 System overview. It follows a layered architecture. The Blockstore layer allows abstract access to fixed size data blocks that can be stored on different storage media. The Blobstore layer uses this interface and offers storing resizeable blobs. The fs++ layer maps the filesystem implementation to different operating systems.

Filesystem The filesystem layer is responsible for storing files and directories using blobs in a blobstore. One file is simply one blob. In a simple implementation, one directory is a list of files stored in one blob, but we also propose alternative implementations; see Section 4.6.

fs++ This is a library we wrote for CryFS. It allows implementing any filesystem against one generic interface and then takes care of the platform specifics and can be used with *libfuse*, *dokan*, and so on. In the current state, only *libfuse* is supported, but adding other native ways to implement filesystems should not be a problem. Especially the kernel variant might be interesting for performance reasons. Thus, fs++ is the layer that achieves the goal of Platform Independence.

libfuse, dokan, Kernel-FS Each operating system has a different way to implement filesystems. Sometimes even multiple ways. *Libfuse* is a way to implement a filesystem in Linux and Macintosh userspace. *Dokan* is something similar for Windows. A filesystem can also be implemented as a kernel module or device driver.

4.4 Encryption Layer

There are some layers in the design where encryption could be implemented. Our choice is to encrypt single blocks in the blockstore when they are written to the disk and decrypt them when they are loaded from the disk. The advantage is that many security properties are inherited to higher layers. To gain any information about the data stored in a block, an attacker has to decrypt the block first.

In the reference implementation, there is one global encryption key for all blocks in the filesystem. This however is the reason it does not achieve the goal of partial shareability. To achieve partial shareability, blocks could be encrypted with individual keys and the used key could be stored at the place where the block is linked from, e.g. at the directory entry pointing to a block containing a file. Since multiple small files are not combined into one block, each block contains at most one file and the user can share keys to single files.

This approach results in a system, where an attacker cannot read the contents of blocks. In a static security scenario, i.e. the attacker cannot make the user modify one blob and see which blocks changed, the distribution of blobs over blocks and the sizes of blobs are also hidden.

4.4.1 Integrity

It is more difficult to achieve the integrity goals. By using an authenticated cipher (e.g. AES-GCM) to encrypt the blocks, it is ensured that an attacker cannot modify single blocks. By storing the block ID in the header of the block, it is ensured that an attacker cannot replace a block with a valid block that had a different ID originally. To prevent an attacker from replacing a block with a previous version of the same block, a block stores a version number in its header. Clients remember the current version number for each block and check that modifications do not decrease it.

The remaining attack vectors are deleting a block, and re-adding a block the user deleted. This is prevented by storing a list of IDs of deleted blocks and check the existing blocks against that list. This list could either be stored locally in each client, or integrity-checked on the server. Both approaches have problems. If the list is stored locally, a client would not be able to accept valid deletions of blocks by other clients. If the list is stored remotely on the server, additional means have to be taken to ensure that a server cannot attack the integrity of this list by for example rolling it back to an earlier version. Furthermore, it would need a way to avoid synchronization conflicts when two clients are deleting blocks at the same time. In the following, we propose two variants to achieve integrity.

Variant 1: Combining Local and Remote Storage

In this variant, each client stores the list of deleted blocks locally, and additionally, it is stored on the server. To prevent synchronization conflicts, the list on the server is stored as one file per deleted blob. There is one directory on the server that contains a lot of small files which each only contain the key of one deleted blob. These files are encrypted and authenticated. Clients check the server side list for updates and update their local list, but they only accept new entries. They do not accept deletions or modifications of entries. This way, a client can delete a block by adding one of these files. An attacker cannot do that, because they cannot fake the authentication of the file. An attacker also cannot undelete a block by removing an entry, because clients do not accept that.

New clients however cannot rely on the server side list, because the server could give them an old version. To ensure integrity when adding new clients to the system, the user gives them a copy of the list via a secure channel. The new client is also given the total number of existing blocks to prevent the server from deleting some before the new client saw them.

Variant 2: Integrity Blocks

Variant 1 has the disadvantage that it needs a lot of small files on the server and also has to keep a potentially large list of deleted block IDs locally in each client. Variant 2 proposes an alternative approach. There is not one small integrity file per deleted block, but there are integrity blocks containing lists of deleted blocks. These integrity blocks are stored like other blocks of the filesystem on the server and are indistinguishable from them. They are encrypted, have a random block ID, a fixed size and therefore also a maximal number of entries. When the maximal number of entries is reached, the client creates a new integrity block. To prevent synchronization conflicts, a client never adds entries to an integrity block created by a different client, but instead creates its own integrity block. The block ID is added to the authenticated header of the integrity block and a version counter ensures that an attacker cannot roll them back. The clients do not need a local copy of the list of deleted blocks, but only need to remember the IDs of the integrity blocks, so an attacker cannot delete them. Although each client has its own set of integrity blocks to write to, they are indistinguishable from normal filesystem blocks, so an attacker cannot match them to clients and does not get any information about the number of connected clients or how many blocks each client deleted.

When adding a new client to the system, it is enough to give them the list of integrity block IDs with their version numbers, and the total number of blocks via a secure channel. A new client does not have to be given the whole list of deleted blocks. If filesystem modifications while adding a new client are prevented, it is also possible to give the new client only a checksum of this integrity data.

Re-Encryption

From time to time, it might make sense to re-encrypt the whole filesystem with a new key. In such a case, the list of deleted blocks can be removed, because all old blocks an attacker could replay do not authenticate with the new key anymore. This might especially make sense when a new client is added, because then the integrity data does not have to be passed to the new client via a secure channel. To avoid having to send the new key to all clients, an indirection can be used. Clients are not initialized with the key to the filesystem, but with the key to a key block that contains a valid key to the filesystem. When re-encrypting the filesystem, first a new valid key is added to the key block, then all blocks are re-encrypted with this new key, and then the original key is deleted. Other clients reading the filesystem during the process have both keys available in the key block. If they are modifying the filesystem, they already use the new key for their changes. This approach also allows encrypting a filesystem with a memorable password instead of a key. The user can decide to encrypt the key block with a memorable password and only has to remember this password. Instead of passing a key file to a new client, a new client can be initialized with that password only and uses it to decrypt the key block and get the key for the filesystem.

Integrity on Blobstore Level

Instead of ensuring integrity on blockstore level, it could also be done on blobstore level by storing a list of deleted blob IDs instead of a list of deleted block IDs. This keeps the lists smaller, because each blob can consist of multiple blocks. However, it makes the system and the integrity proofs more complicated, because it does not prevent an attacker from rolling back single blocks. So it has to be ensured that an attacker cannot modify parts of a blob by rolling back single blocks. Furthermore, the system is easier to understand and to check for correctness when all security related code is in the same layer. Since an attacker is prevented from modifying blocks because an authenticated cipher like AES-GCM is used in the blockstore layer, it makes sense to have the remaining integrity code in the same layer and not in the blobstore layer.

Conclusion

It is easier to ensure integrity on the blockstore level. Comparing the two variants, there is a disadvantage of the second variant that when a new block is synchronized to a client, that client has to check it for whether it is an integrity block, and in case add it to its list. This is a small performance disadvantage, but we think the performance advantage of not having many small integrity files as in variant 1 is larger. Furthermore, the second variant makes it easier to add new clients, because they do not need the whole list of deleted blocks. In CryFS, integrity is ensured as described in variant 2.

4.5 Blobstore Layer

The blobstore layer is responsible for mapping resizeable blobs onto a blockstore where all blocks have the same size. This is done using balanced left-max-data trees as described in Section 2. Each tree node is stored in a block and the leaf blocks contain actual data.

Since blobs represent files in the filesystem, there has to be a fast way for random read and write accesses and resizing blobs. The corresponding algorithms are described in Sections 2.9 and 2.10.

An alternative to storing the inner tree nodes in blocks is to keep the inner nodes in the client only. However, a new client then has to rebuild the tree structure from the leaf blocks, which can be a costly operation. This is why the reference implementation stores the inner nodes as actual blocks on the server. Section 2.8 shows that the space overhead for this is very low.

If a blob is smaller than one block, the space overhead is larger, because the system needs to allocate at least one block for each blob. To avoid wasting space for small blobs, small blobs could be merged into one block. This however results in synchronization conflicts when two clients are modifying two different blobs that happen to be in the same block. This is why CryFS does not merge small blobs into one block.

As described in Section 4.4, the blockstore already guarantees integrity and confidentiality for the whole blockstore; i.e. an attacker cannot get information about, modify, delete, add or rollback blocks. Since all data in the blobs is stored in blocks, these security properties are directly inherited for the blobstore without additional effort. This is proven in Section 7.

4.6 Filesystem Layer: Storing Directory Structure

A blobstore allows to store arbitrarily resizeable blobs of data. Mapping files onto that structure is simple. Each file is stored in one blob. Mapping directories needs some more thought and there are many possible designs.

The following goals are a subset of the goals in Section 4.2 and are especially important when designing a way to store directory structure. In this analysis, the goal of Local Performance is divided into two separate subgoals, Fast Path Lookups and Fast Modifications, because these subgoals are often conflicting and designs have to make a trade off.

Fast Path Lookups	When the user accesses a file, the system gets its path but needs its blob ID. This lookup should be fast, even if the file is deeply nested.
Fast Modifications	Renaming and moving directories and files should be fast.
Synchronization Compatibility	When two clients modify the same directory (for example both add a file), third party synchronization clients can easily resolve this conflict. In the case of CryFS, because directory structure is mapped to blocks, this is not so simple and needs special thought. Furthermore, the system should be able to handle data that is incompletely synchronized.
Storage Efficiency	The system should use as little space as possible.
Structural Confidentiality	An adversary should not get any information about the size of individual files or about the directory structure.
Structural Integrity	An adversary should not be able to manipulate file sizes or the directory structure without the user noticing it.
Partial Shareability	A user should be able to share a subset of files/folders with a friend without giving them access to other files.

With these goals in mind, we explain some of the designs and discuss their advantages and disadvantages in the following sections.

4.6.1 Central Directory Structure

A first design approach is to keep directory structure in a central place; i.e. in one central blob. Figure 4.3 illustrates the concept. This allows fast lookups and also fast renaming and moving, because all changes happen in that one central blob. The approach has some confidentiality risks though. If an attacker can make the user move some files and look at which blocks changed, they can easily figure out which blocks store this central directory structure blob. Having that, the advantage of distributing this blob over multiple same-size blocks is lost and it could just as well have been stored in one dynamically sized block. This might not be a problem if the system could make sure that the directory structure does not impact the size of this blob, but that is very difficult. A filesystem containing a lot of small files needs a lot more entries than a filesystem containing few big files. Filename lengths and the depth of the directory tree are also likely to cause differences in the directory blob

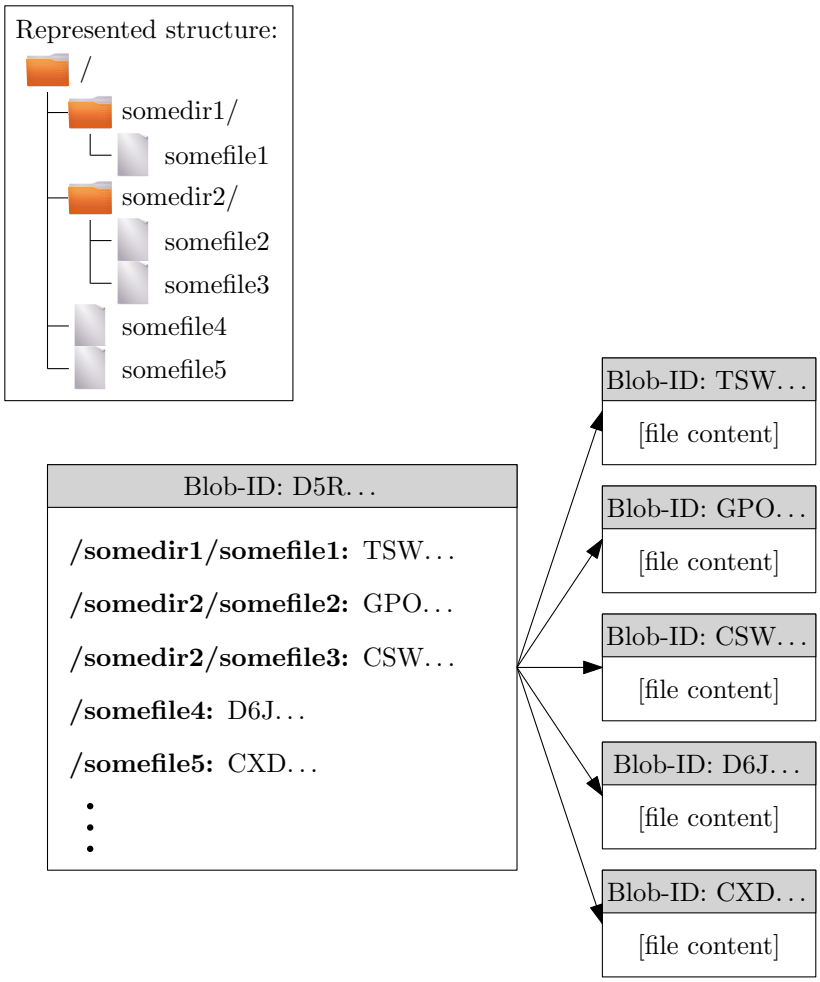


Figure 4.3 Example of how directories are stored in a central directory structure. The blob ID for the central directory structure is "D5R...". This blob contains a list of all files in the filesystem, each entry pointing to the ID of the blob containing this file.

size. By seeing the size of the directory blob, an attacker could distinguish these filesystems. To prevent that, the system could allocate a max-size directory blob in advance, but that would waste space and introduce a maximal filesystem size. Another disadvantage of this basic variant is that there is one central blob (few central blocks) which is modified very often. This results in synchronization conflicts happening often, even when two clients modify different directories.

Variant: One Blob per Client

A variant solving the problem of synchronization conflicts is to store one central directory structure blob per client. Clients can only write to their own copy, but monitor the other ones. This way, conflicts do not arise in the third party synchronization tool (which would probably just let one version survive), but the client actually gets both versions and can resolve the conflict. When a client notices that another client has an additional entry, it has to decide whether the other client added this entry or whether the client itself deleted

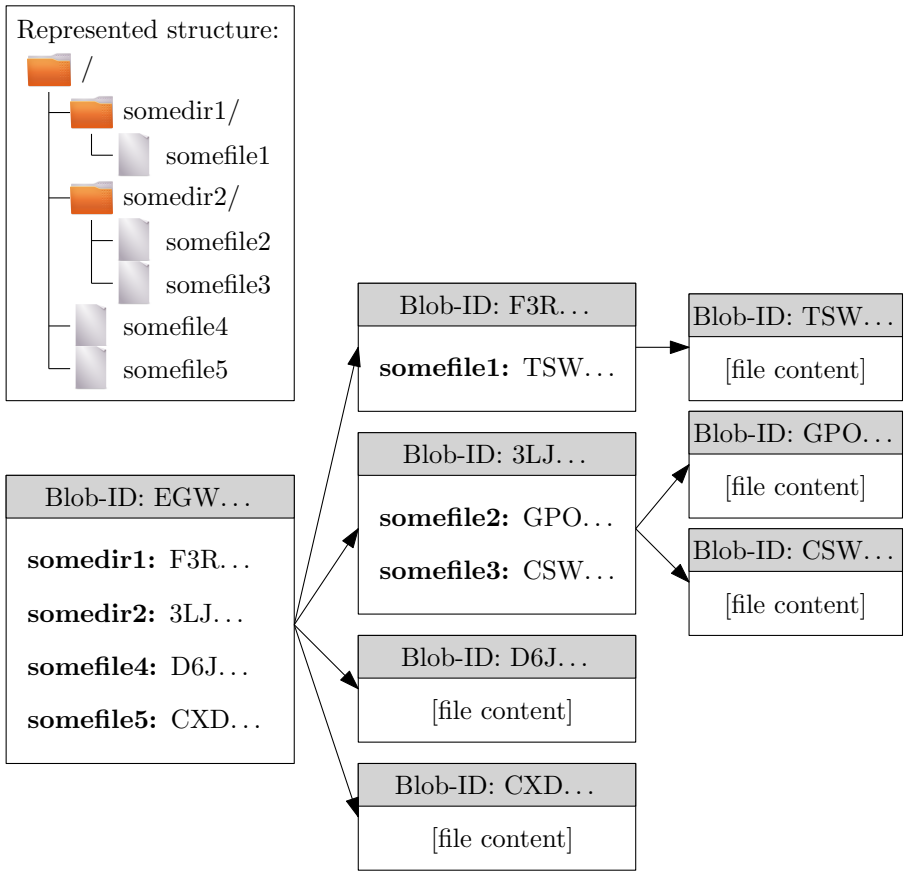


Figure 4.4 Example of how directories are stored in a directory blob structure. The blob ID for the root directory is "EGW...". It is a directory blob storing the direct entries of the root directory with their name and the ID of the blob where the entry is stored. The blobs containing these entries are either directory blobs as well, or store file contents.

it. One approach for this is to remember deleted files until the directory structure blobs of all other clients adopted to the change, another approach is to use version numbers. A client could also look at whether a blob actually exists to decide whether it was added or deleted. The confidentiality issue of the base variant also affects this variant.

Conclusion

Although the proposed variant solves the problem with synchronization conflicts, it is a complex solution. This together with the confidentiality risks mentioned cause that it is not used by CryFS.

4.6.2 Directory Blobs

A second design is to store a directory as a list of entries in a blob; see Figure 4.4. Each file blob and each directory blob has a random blob ID. Each directory blob contains a map from the name of the entry to the blob ID where this entry is stored. A root directory blob stores the entries of the root directory. To allow fast entry lookups, this map can either be

a hash map, sorted list, or any other datastructure allowing fast lookups. Path lookups are slower, because the client has to traverse and decrypt all directory blobs in the path. Modifications are fast, because renaming or moving a large directory only needs modifying the entries in the old and the new parent directory blob. To allow partial shareability, each blob is encrypted with its own key and this key is stored together with the blob ID in the parent directory blob. Giving someone the key to an arbitrary directory recursively allows them to decrypt the key to all entries, so they get access to the whole directory subtree.

In the basic variant, this approach cannot handle multiple clients modifying the same directory. If two clients modify the same directory, only one of the versions survives the conflict. The client however sees orphaned blobs in this case. If there is a parent pointer in the header of each blob, clients could resolve synchronization conflicts by re-adding orphaned blocks to their directories. To fulfill the integrity goal, this requires the blobstore to ensure that an attacker cannot re-add a deleted blob as described in Section 4.4.1. However, if the system uses parent pointers to resolve conflicts, the direct parent pointer approach in Section 4.6.5 is actually better.

The following variants discuss solutions to the disadvantages of the directory blob design, namely slow path lookups and synchronization conflicts.

Variant 1: Encoding Paths into Blob IDs

A variant achieving fast nested lookups is to make blob IDs not random but deterministic, namely the encrypted hash of the file path. If the client wants to access a file and has its path given, it can quickly get its blob ID by hashing and encrypting the path. The disadvantage of this solution is that modifications become slow. When renaming or moving a large directory, the system has to give all entries in this subtree a new blob ID, because their path changed.

Variant 2: Dummy Entries

A variant partly solving the problem with synchronization conflicts is to give directory blobs a lot of dummy entries, pointing to nonexistent blobs. The directory blob does not store the entry name itself, only the child blob stores the name. If a client adds an entry to a directory, it simply chooses one of the dummy entries and creates a blob with that blob ID, making it a real entry. This does not need to modify the directory blob itself. Deleting an entry also does not need to modify the directory blob itself, because the blob can simply be deleted and the directory entry becomes a dummy entry. Since the entry name is not stored in the directory blob, renaming also does not modify the directory blob. Although this solves most of the conflict scenarios, there are some remaining. There is a chance that two clients adding a file to a directory choose the same dummy entry. To make this chance low, a directory would have to have a lot of dummy entries, making it waste a lot of space. There is also the possibility of conflicts when a client grows a directory blob to add new dummy entries because a directory got too large. Clients have to decide about growing the directory blob indeterministically to avoid two clients doing that at the same time. Another disadvantage of this approach is that listing directory content is slow, because the filesystem has to open and decrypt all directory entries to get their names.

Variant 3: Implicit Dummy Entries

Taking the last variant further, directory blobs do not actually need to be stored anymore. Blob IDs can be chosen in a deterministic way, namely as an encrypted hash of [directory-ID]:[entryindex], being an implicit list representation without explicitly storing it. Clients adding files choose a random free entry index when adding an entry to a directory. Deleting an entry can be done by simply deleting the entry blob. This approach has fewer conflicts, but makes listing directories even slower, because the client has to check all possible entry indices for whether they exist. This is again a trade-off between probability of conflicts and performance. To get few conflicts, clients that are adding an entry should be able to choose from a large range of possible entry indices.

It also makes partial shareability more complicated, because there are no directory blobs where encryption keys for directory entries could be stored. Furthermore, this requires the blobstore to ensure that an attacker cannot delete blobs or re-add deleted blobs, otherwise they could manipulate the directory structure. Such a blobstore is described in Section 4.4.1. The other variants also need a mechanism to prevent rollbacks, but they do not allow the attacker to put a directory into a state that never existed in the past. However, if the blobstore does not prevent this, then this variant allows the attacker to arbitrarily delete single files or add a subset of old files without the user noticing.

Variant 4: In-Memory Index

A variant combining fast path lookups and fast renaming and moving is to store directory blobs like in the basic variant, but to keep a local in-memory map from paths to blob IDs in each client. To notice changes, clients have to periodically rebuild this index, which costs performance. There is a way to use incremental rebuilds; see the In-Memory Index variant of Path Headers in Section 4.6.4.

Variant 5: Cache

This is similar to the last variant, but instead of a full index, a last-recently-used cache is used. Entries expire after a short timeout to avoid synchronization conflicts. When accessing the same file multiple times, it is already in the cache. When accessing many files in a nested directory, there still is a benefit because the client does not have to lookup the blob ID of the directory multiple times. This variant does not solve the problem with synchronization conflicts. Two clients modifying the same directory still leads to conflicts.

Conclusion

In a directory blob design, partial shareability is easy to implement. Using an in-memory index or a cache, the design allows for fast path lookups. We prefer a cache to an in-memory index, because it is easier to implement, does not have a long bootup time, does not need periodical or incremental rebuilds and uses less memory. Synchronization conflicts remain a problem, since the proposed dummy entries variants are not perfect solutions and have their own problems. However, they are only a problem if two clients modify the same directory. A directory blob design is very easy to implement. This is why the current implementation of CryFS uses it. We plan to switch to the parent pointer design introduced below in later versions.

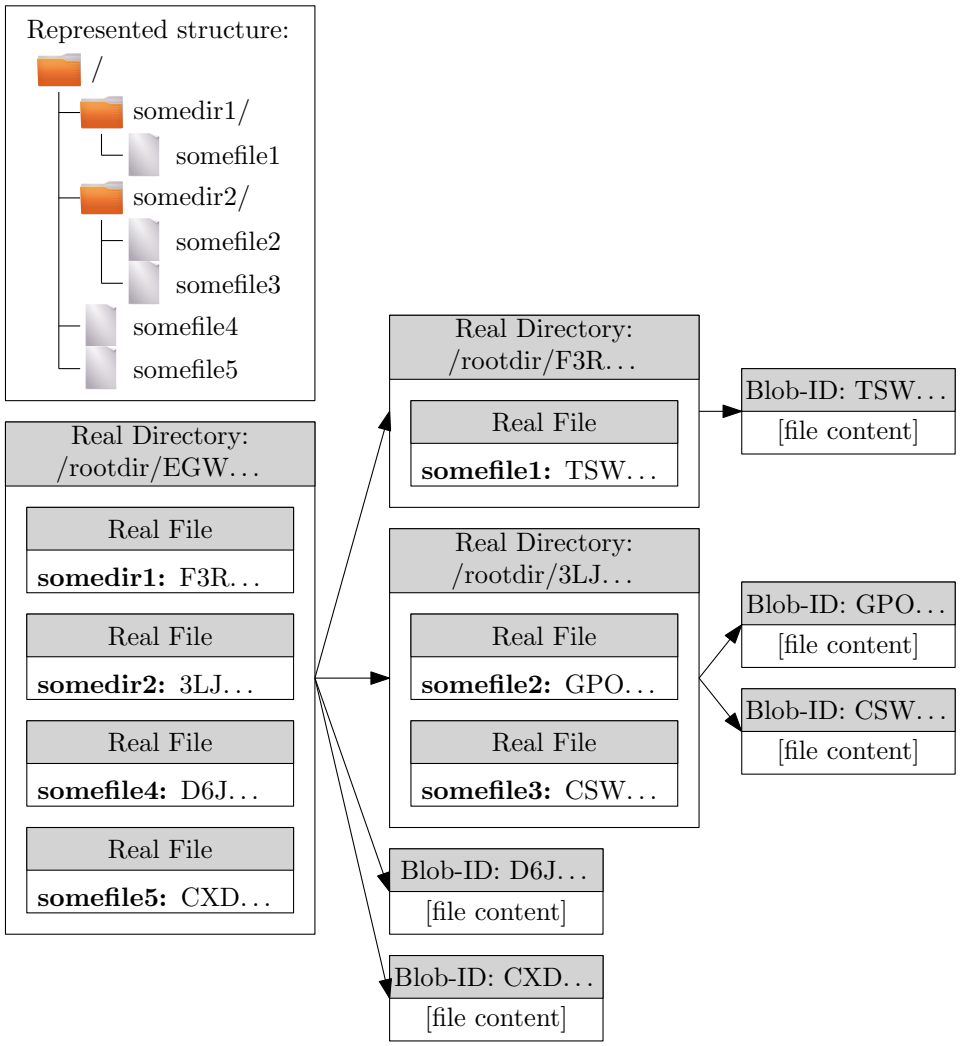


Figure 4.5 Example of how directories are stored in a real directory structure. All real directories are on top level under /rootdir/ and each real directory represents a (possibly nested) virtual directory in the plaintext filesystem. The root directory is stored in "/rootdir/EGW...". Each real directory contains real files, one file per entry. An entry contains the corresponding file or directory name and an ID as a pointer to where the entry is stored. In the case of a file entry, this ID is the blob ID. In the case of a directory entry, this ID points to another real directory containing the entries of this subdirectory.

4.6.3 Real Directories

The basic idea behind this approach is that third party synchronization clients already solve the problem of avoiding conflicts when multiple clients are modifying the same directory. CryFS can use their solution instead of implementing an own one. To use their solution, directories have to be represented as real directories in the underlying filesystem, the blockstore abstraction cannot be used for them. However, if done like described in the following, it still fulfills the security goals.

As before, each directory in the filesystem has a blob ID and stores a list of blob IDs as entries. If the entries were stored together in one file, synchronization conflicts could not be avoided. So each entry is stored in its own file. A directory is now represented as a real directory, whose name is the directory blob ID, and this directory contains a lot of very small entry files, each file only containing the name and the encrypted blob ID of the entry; see Figure 4.5. These real directories are all on top level, there are no nested real directories. Instead of storing the encrypted blob ID in the content of the entry file, it can be used as the entry file's name, which on most operating systems makes lookups faster. It has to be encrypted, because an attacker should not be able to see which blobs are the entries of this directory, since that would allow reconstruction of the directory tree. The filenames of the entries are not stored in the file blobs, but in these small entry files to allow faster directory listing. However, clients still have to access one file per entry, so listing a directory in this design is slow.

So far, this design does not ensure integrity. An attacker can easily remove entries, replace entries, or add old entries. That can be solved using an integrity list similar to Section 4.4.1.

Variant 1: Dummy Entries

The base variant has the problem that an attacker can see how many directories there are and how many entries each directory has. To solve the first problem, dummy directories can be introduced. To solve the second problem, the number of entries in a directory can be fixed. Each directory has exactly n entries, of which at most m are real entries and at least $n - m$ are dummy entries. When a client adds a new entry to a directory, it replaces one of the dummy entries with a real entry. Because it does not change the dummy entry but deletes it and adds a new entry instead, it avoids synchronization conflicts when two clients are adding an entry to the same directory. The only conflict that can happen is that two clients choose the same dummy entry to delete, in which case there is one entry too much in the directory after both clients finished. This does not hurt correctness. An attacker can see this way that there are at least two clients working with the directory, but this is not necessarily something the system wants to hide. If it wants to hide it and an attacker is not able to see the intermediate state of the filesystem, then the client can delete a dummy entry whenever it notices there is one too many. Each client periodically and randomly decides whether they want to fix the number of dummy entries. If two clients decide to fix the same issue, there is be a dummy entry too few afterwards and the clients can use the same algorithm to add a dummy entry again.

Because the maximal number of directory entries should not be limited, the system uses link entries. If a directory has close to m entries, a client can add an indirect entry pointing to another real directory where the list of entries continues. When two clients decide at the same time that they have to do this, there are multiple link entries in the directory, which does not hurt correctness and also does not give an attacker more information. For very large directories, it might even be desirable to have such a tree structure instead of a long linked list.

So with this variant, an attacker cannot see the number of directories or the number of their entries anymore. However, even with dummy directories, they can still distinguish between the data that stores directories and the data that stores files. Using this, they might be able to distinguish a filesystem layout where all files are in the root directory from a very nested filesystem layout with a lot of directories.

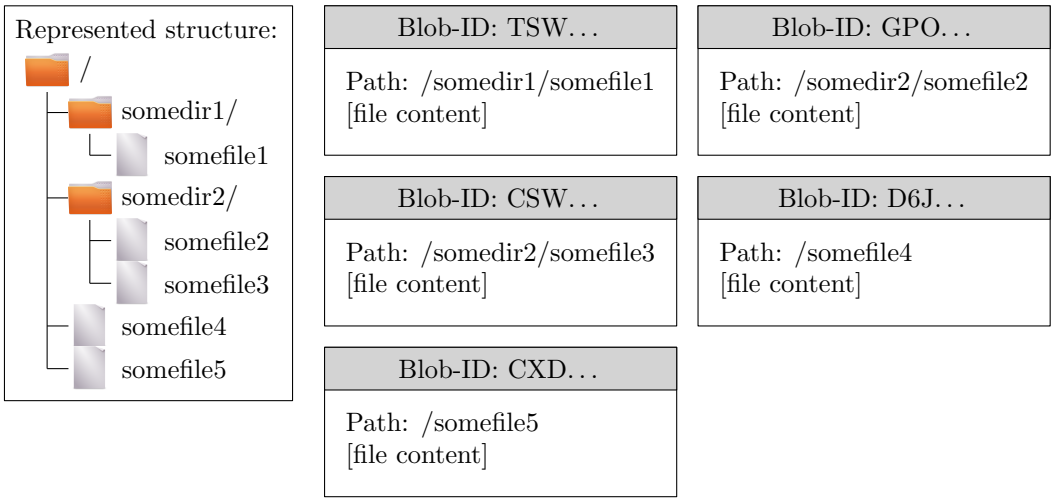


Figure 4.6 Example of how directories are stored in a path header structure. Directories are not stored, but only file blobs. Each file blob stores its path in the header. Clients can build a local directory index by scanning over all headers.

Variant 2: Storing Files and Directories Together

To avoid an attacker being able to distinguish between files and directories, real directories can be merged with file blocks. Instead of storing blocks as files that are independent from this directory structure, there is one block stored in each real directory. So each real directory stores a fixed number of small files for directory entries, and additionally, it stores a larger file which contains one block of file data. If this block is referenced as a directory, the client only accesses the small files and reads the entries. If this block is accessed as a file block, the client only reads the larger file.

With this variant, an attacker cannot distinguish between directory data and file data anymore. However, if a block contains only directory entries or only file data, the system also has to store the other. That is, if the number of total directory entries and the number of file blocks has a different ratio than what we expected when designing the system, the system wastes a lot of space.

Conclusion

In a real directory design, listing directory contents—and therefore also path lookups—are slow. Path lookups could be made fast using a cache, but this does not help for listing directory contents, because they can be large. If an attacker should be prevented from seeing the number of directories or the number of their entries (variant 1 or variant 2), the design gets complex and possibly wastes a lot of space. If this is not done, then an attacker can not get the actual directory tree, but still distinguish two filesystems with a different directory tree. In conclusion, although it sounded like a good idea to have the third party synchronization client handle synchronization conflicts, this design is not preferable.

4.6.4 Path Headers

In the path header design, each file blob stores its entire path in the header of its blob; see Figure 4.6. With this approach, renaming and moving large directories takes a lot of

time, because the operation has to change the headers of all file blobs in the directory subtree. Path lookups can be made fast by keeping additional data like an in-memory index or a cache. This design also requires the blobstore to ensure that an attacker cannot delete blobs or re-insert deleted blobs, because otherwise an attacker can not only roll back the whole filesystem, but also delete files or re-insert deleted files to bring the filesystem into a state it has never been in. A blobstore implementing this integrity is described in Section 4.4.1

Variant 1: In-Memory Index

In this variant, each client keeps a local cache of path to blob ID mappings. This avoids synchronization conflicts and allows fast path lookups, but for building this index, clients have to scan all blobs, which potentially causes a long bootup time. Furthermore, the client has to keep the index up to date. To avoid periodically scanning all blobs again, an invariant could be introduced that each change in directory structure causes new blobs to be created or old blobs to be deleted. This is trivially fulfilled when adding or deleting files. When a client renames or moves a file, it has to give it a new blob ID. Given this invariant, the client does not have to periodically scan all blobs, but it only has to keep a list of blobs it knows and check for added or deleted blobs.

With or without this invariant, blocks could store a boolean flag in their header, specifying whether they are the root node of a blob. That makes it faster to scan over all blocks and find and recognize the blobs. So to build the in-memory index, a client could scan over all blocks, and if it finds a root node, it traverses down to its first leaf to read the blob header, which contains the path.

This traversing down the tree still needs some time and could be made faster by storing the first few bytes of a blob in the root node itself. Then, the chance would be quite high that the path is also stored in the root node itself and the client does not have to traverse down to the first leaf. Alternatively, the root node could contain a direct link to its first leaf so at least the client does not have to traverse the whole tree depth down. Another idea is to not mark the root node but the first leaf with a flag. But then, the blob header contained in the first leaf has to store the blob ID, because (a) it is needed to build the in-memory index and (b) for very long paths, the second leaf has to be accessed.

Variant 2: Directory Blobs with Cache

In this variant, the system stores directory blobs additionally to the path in the blob header. The directory blob variant with a cache is used to keep nested lookup times fast. This variant avoids a long bootup time, but introduces the possibility for synchronization conflicts in the directory blobs. However, if a synchronization conflict happens, clients could always recover from it and rebuild the directory blobs from the file blob headers.

Variant 3: Central Directory Structure

Instead of using directory blobs, the file blob header approach could also be combined with a central directory structure. This variant also avoids a long boot time. It also introduces the possibility for synchronization conflicts, which can also be recovered by rebuilding the central directory structure from the file blob headers. However, using directory blobs is

still a better idea, because it does not have some few blocks that are accessed on each modification and therefore synchronization conflicts happen less often. Additionally, a central directory structure introduces the confidentiality issue that an attacker can get information about the filesystem by looking at the blob containing the central directory structure; see Section 4.6.1.

Conclusion

In a path header approach, we need the blobstore to ensure that an attacker cannot delete or re-insert blobs. However, this is fulfilled by the CryFS blobstore. Path lookups can be made fast by using an in-memory index or a cache. We prefer a cache to an in-memory index, because it is easier to implement, does not need periodical or incremental rebuilds and uses less memory. When using the variant with directory blobs, the long bootup time is avoided and partial shareability gets possible. Renaming and moving large directories is slow. This can be solved by storing parent pointers instead of the file paths in the blob headers, as described in the following section.

4.6.5 Parent Pointers

This design is similar to the path headers design, but instead of the full path, the system only stores the name and a parent pointer in the blob header; see Figure 4.7. The parent pointer is the ID of the parent directory blob. Directory blobs exist, but they only contain the directory name and a parent pointer. They do not store their entries. The advantage over the path header approach is that renaming and moving large nested directories is fast, because the operation only has to modify the name or parent pointer of the directory itself and does not have to look at its entries. Like in the path header approach, nested lookups can be made fast by keeping an in-memory index, storing entries in directory blobs or having a central directory structure. The parent pointer design also inherits the need for relying on the blobstore doing additional effort to ensure integrity from the path header approach. Regarding the goals, the parent pointer design is strictly better than the path header design, because it has all its advantages and one disadvantage less.

Allowing Partial Shareability in a Parent Pointer or Path Header Approach

There are two degrees of partial shareability. In an optimal solution, the user is able to share single files or whole directories. This is the level that is supported by the directory blob approach. A weaker degree of partial shareability is to allow sharing whole directories only. In the following, we describe how to implement whole directory sharing in a parent pointer or path header approach.

In a parent pointer approach, there still are directory blobs, but they only store a parent pointer and do not store their entries. These directory blobs could be used to store the unique key that has been used to encrypt the entries in this directory. In a path header approach, such directory blobs could be introduced. When a client boots up, it only knows the key for the root directory. From the root directory blob, it gets the keys for the direct children, but not more. So it has to try each block with each known key in an iterative process to read the whole directory structure. This could take quite a long bootup time. If the parent pointer or path header design is implemented in the directory blob variant, where directory blobs also store their entries, then the bootup time is much faster.

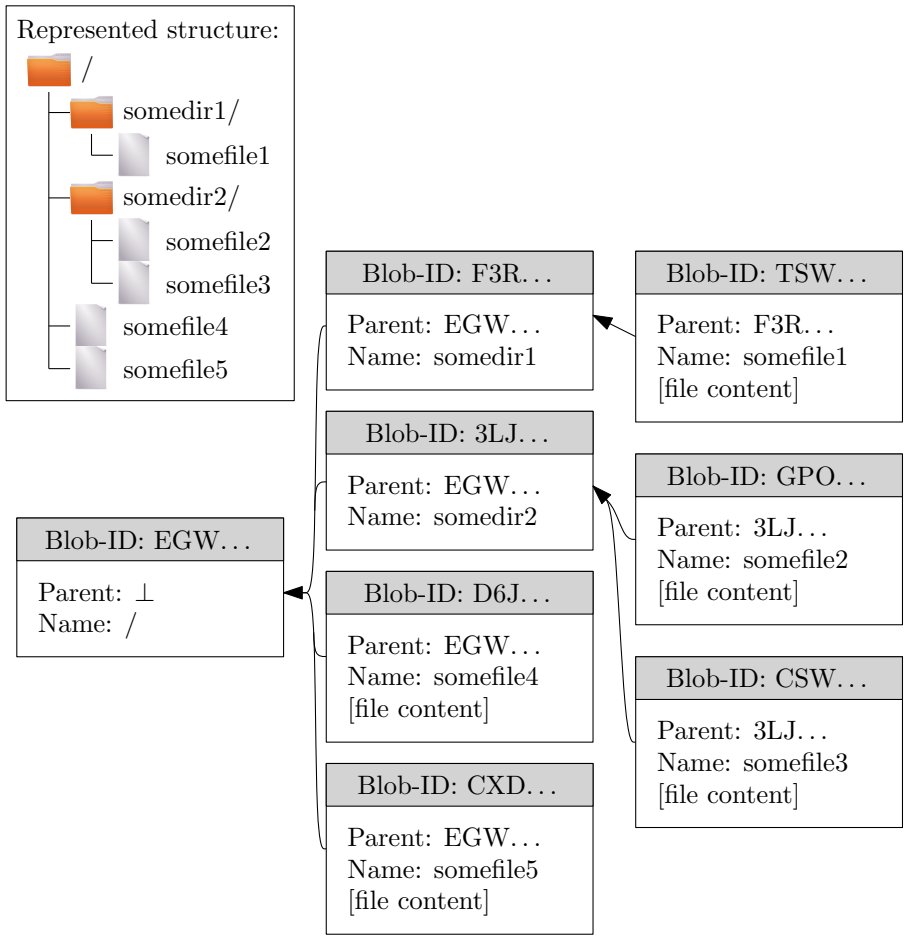


Figure 4.7 Example of how directories are stored in a parent pointer structure. Each file is stored in a blob and stores its name and a parent ID as a pointer to its parent directory. Directories do not store their entries, but only their own name and a parent pointer. Clients can build a local directory index by scanning over all blobs. The root directory is stored in "EGW...".

Another approach to partial shareability that also allows to share single files is on-demand re-encryption. All files are encrypted with the same key at first. When sharing a folder, a client re-encrypts this part with a second key and shares the second key. It also remembers locally that this part is encrypted with a different key, so it itself is able to continue accessing it. This is an approach that also allows un-sharing a file by re-encrypting it with the first key again. The actual implementation is quite complex though. If the client shares a folder with person A and then afterwards wants to share a subfolder with person B, then also person A has to be notified and has to get both keys.

Attribute Based Encryption [Goy+06; HW13; LCH13] could be a solution to partial shareability, but known implementations for attribute based encryption are quite slow. It might be possible however to give each file a unique encryption key and use attribute based encryption only to encrypt and store these keys. This is left for future work.

Legend

++ Fully fulfilled

\+ Fulfilled but needs additional effort

– Moderately fulfilled

– – Not fulfilled

	Fast Path Lookups	Fast Modifications	Synchronization Compatibility	Storage Efficiency	Structural Confidentiality	Structural Integrity	Partial Shareability
Central Directory Structure	++	++	+	++	–	++	+
Directory Blobs	+	++	–	++	++	++	++
Real Directories	+	+	++	+	+	+	+
Path Headers	+	– –	++	++	++	++	+
Parent Pointers	+	++	++	++	++	++	+

Figure 4.8 Different design choices for storing the directory structure and the goals fulfilled by them. A parent pointer design is the best choice. With a bit additional effort for fast path lookups and partial shareability, it can fulfill all of the goals.

4.6.6 Conclusion

Figure 4.8 shows an overview of the design alternatives and which goals they fulfill.

A central directory structure has problems with structural confidentiality and real directories need a lot of effort to implement them with structural confidentiality. The path header approach is strictly dominated by the parent pointer approach. We think a parent pointer approach in the variant using directory blobs with a cache is the best solution for representing directory structure. It fulfills most of the goals. It allows fast nested lookups, fast renaming and moving and avoids synchronization conflicts. Although the plain parent pointer design does not support partial shareability without getting the disadvantage of a long bootup time, the parent pointer design in the directory blob variant solves this problem. The parent pointer approach is however more complicated to implement, which is the reason the reference implementation uses a plain directory blob approach. We are planning to implement the parent pointer approach in a future version.

5. System Reference

In this section, we define the storage layout for CryFS. The system is described on a higher level in Section 4, whereas we describe the implementation details here.

5.1 Config File

The configuration file is kept locally on the client. Some other encrypted filesystems like EncFS keep it on the server, but that allows attackers to change it and in that specific case switch off integrity checks. Figure 5.1 shows an example configuration file. It is stored in JSON format and contains the following settings:

rootblob The ID of the blob containing the root directory.

key The cryptographic key used for encryption, stored as a hexadecimal value.

cipher The cryptographic cipher used, e.g. "aes-256-gcm".

Supported ciphers are aes-{128,256}-{gcm,cfb}, twofish-{128,256}-{gcm,cfb}, serpent-{128,256}-{gcm,cfb}, cast-256-{gcm,cfb} and mars-{128,256,448}-{gcm,cfb}.

```
{
  "cryfs": {
    "rootblob": "7C426C036F5A3CCCD7801365CFAEE4C8",
    "key": "D47AADF4112EDFE48D87691ADBE1265253A056DD7CF806463BD7BF705B4F0426",
    "cipher": "aes-256-gcm"
  }
}
```

Figure 5.1 A CryFS config file. The rootblob variable stores the blob ID for the root directory as an entry point for the filesystem. The configuration file also stores the cryptographic cipher and a hexadecimal representation of the cryptographic key.

16 bytes	block ID	The ID of the block. This prevents attackers from replacing a block with a different block.
uint8_t	depth	The depth of the subtree with this node as root node. For leaves, this is 0.
3 bytes	not used	unused space for byte-alignment (might be used in future versions).
uint32_t	size	For internal nodes: number of child nodes. For leaves: number of bytes stored.
remaining	data	For internal nodes: pointers to child nodes. Each pointer stores the block ID of the child node and uses 16 byte. For leaves: data bytes.

Figure 5.2 Layout of a decrypted block in the tree.

uint8_t	magic number	For directory blobs: 0x00. For file blobs: 0x01. For symlink blobs: 0x02.
remaining	data	For directory blobs: Directory entries. Each entry has the layout shown in Figure 5.4. For file blobs: file content. For symlink blobs: Target path for the symlink. The blob-size terminates the string, there is no null byte at the end.

Figure 5.3 Layout of a blob.

uint8_t	child type	For directories: 0x00. For files: 0x01. For symlinks: 0x02.
null-terminated	child name	Name of the entry.
null-terminated	child ID	Blob ID of the entry.
uid_t	uid	POSIX user id flag.
gid_t	gid	POSIX group id flag.
mode_t	mode	POSIX mode flag, i.e. permission bits.

Figure 5.4 Layout of an entry in a directory blob.

The GCM ciphers support integrity, while the CFB ciphers are meant for use cases where integrity is not needed. Future versions will also support CBC, CTR, CCM and EAX modes. CCM and EAX are combinations of CTR mode with CBC-MAC respective OMAC for integrity.

5.2 Block Layout

Each block stores one node of a balanced left-max-data tree and is stored as a real file. The filename is the block ID. Block IDs are 16 bytes long, represented as a hexadecimal string with 32 characters.

Each block is encrypted with the configured encryption scheme. The initialization vector is prepended to the encrypted data.

When it is decrypted, a block has the layout shown in Figure 5.2.

5.3 Blob Layout

Each blob is one balanced left-max-data tree. The inner node blocks define the order of the leaf blocks and allow fast accesses and modifications. The ordered leaf blocks store the data of the blob.

A blob can either store a file, a directory, or a symlink and has the layout shown in Figure 5.3. In case of a directory blob, the layout of a directory entry is shown in Figure 5.4.

6. Implementation and Evaluation

In this section, we explain the software architecture and design decisions of the reference implementation. We also provide performance experiments and show that it is fast enough to be used in practice.

CryFS is implemented using C++ for performance reasons and can be compiled with either GCC or Clang. For cryptography, the Crypto++[1] library is used. It is written in a way that makes it simple to swap this library for another one, e.g. openssl[2]. The code is well covered by test cases to help robustness and stability.

6.1 Software Architecture

The top level architecture follows the top level system design as described in Section 4.3. Figure 6.1 shows the main components.

The fs++ layer is a library we wrote for CryFS that allows the implementation of a filesystem against a portable interface. This enables the filesystem to run on different platforms using libfuse on Linux or Macintosh and dokan on Windows. On the other end of the layer stack, a blockstore can work on any storage backend. The most common case is storing blocks in the local filesystem (and maybe synchronize them using third party clients), but it is easy to implement a storage backend for NFS or S3.

6.1.1 Blockstore

There is a lot of functionality built into this layer. Each functionality is put into its own sublayer. All sublayers are implementing the same blockstore interface, only accessing the blockstore layer directly below and providing some additional functionality to the layer above. Each request passes through all the sublayers. The sublayers are shown in Figure 6.2.

[1] https://www.cryptopp.com/
[2] https://www.openssl.org/

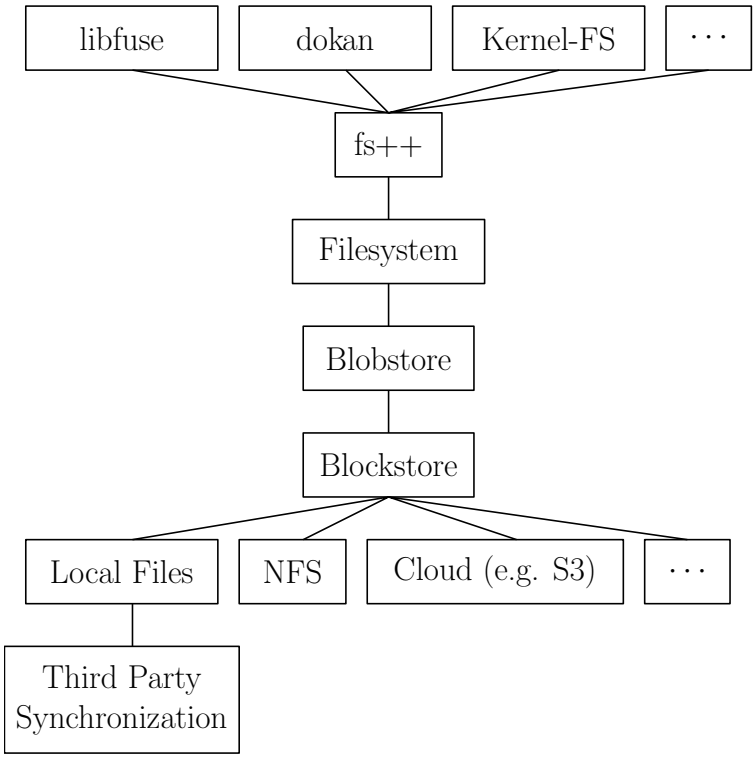


Figure 6.1 Top level architecture. The Blockstore layer allows abstract access to fixed size data blocks that can be stored on different storage media. The Blobstore layer uses this interface and offers storing resizeable blobs. The fs++ layer maps the filesystem implementation to different operating systems.

OnDiskBlockStore

This blockstore works on local files. It implements the blockstore interface and stores each block in its own file in a common directory. The filenames are the block IDs. When there are too many files in the directory and directory lookups become slow, it groups them in subdirectories by the first few bytes of the block ID.

EncryptedBlockStore

This blockstore works on a base blockstore and adds encryption and integrity checks to it. A user of this layer works with plaintext data, and this layer then stores the ciphertexts in the base blockstore.

CachingBlockStore

This blockstore also works on a base blockstore and adds caching functionality. When a block is returned by the application to this blockstore, it does not return it to the base blockstore yet, but keeps it in a cache. If the application then requests the same block again, it delivers it from the cache. Blocks are returned from the cache to the base blockstore after a timeout or if the maximal cache size is exceeded. In the reference implementation, a timeout value of 1 second is used, because it ensures fast synchronization and still has a large performance gain. The performance impact is mostly due to the way

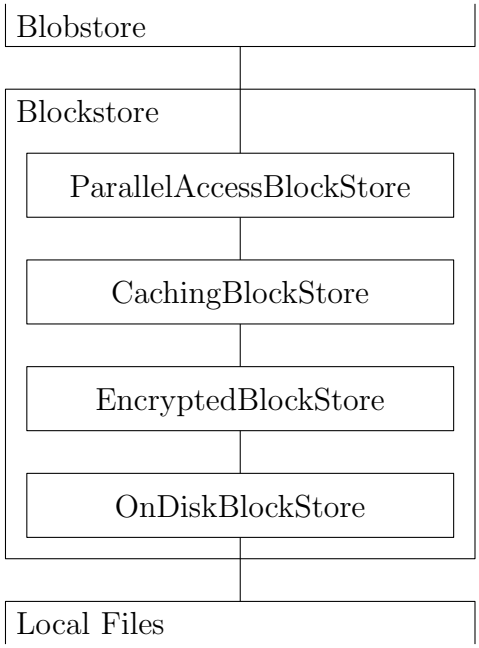


Figure 6.2 Blockstore Architecture. It is divided into sublayers. The OnDiskBlockStore offers a blockstore interface that stores the blocks on a local hard disk. EncryptedBlockStore and CachingBlockStore add a cryptography layer, respective a caching layer to that. ParallelAccessBlockStore deals with race conditions when accessing the same block at the same time in different threads.

libfuse is implemented on the operating system side. When an application sends a large read/write request to a libfuse filesystem, the operation is split into suboperations, each reading/writing a small block. The CryFS implementation then gets a read/write request per suboperation and can work on them in parallel. However, these block sizes are not necessarily aligned with the block size of CryFS. This is why many of these suboperations have to access two CryFS blocks and each CryFS block is accessed by two suboperations. The cache takes care that this double access happens in memory and CryFS does not have to write blocks back to the disk just to immediately reload them. This caching layer is on top of the encryption layer, because performance is better when keeping encryption operations to a minimum.

ParallelAccessBlockStore

A blockstore interface offers the application to load blocks, perform read/write operations, and then write them back. If an application loads the same block twice and writes both back, then the last block wins and the other change is lost. This race condition is solved by this layer. It also works on a base blockstore and basically only passes through blocks from this base blockstore, but if the same block is requested twice, then it does not return two distinct block objects, but the same block object to both. Since the block objects are implemented threadsafe, the race condition is solved.

6.1.2 Blobstore

This layer is also separated into sublayers as shown in Figure 6.3.

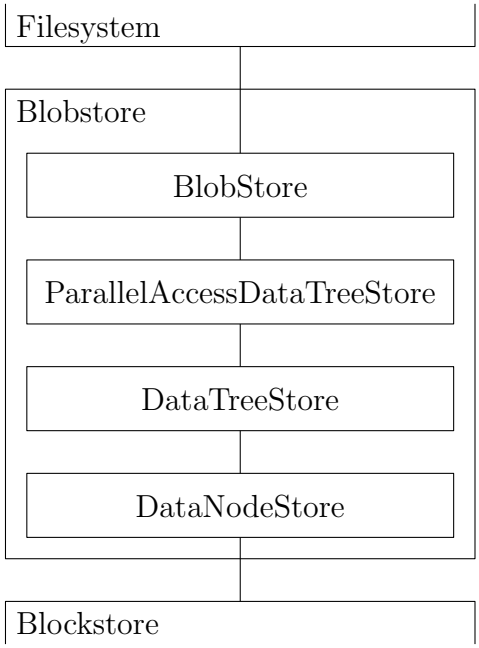


Figure 6.3 Blobstore Architecture. It is divided into sublayers. DataNodeStore offers abstract access to nodes of a balanced left-max-data tree, each node stored as a block in the underlying Blockstore. DataTreeStore adds tree algorithms and offers access to whole trees. ParallelAccessDataTreeStore handles race conditions when accessing the same blob in different threads and the top layer is an adapter that offers the BlobStore interface.

DataNodeStore

This layer works directly on a blockstore, but instead of offering blocks, it offers loading tree node objects. Internally, it maps the tree nodes to blocks and stores them in the blockstore.

DataTreeStore

This layer is responsible for combining tree nodes to whole trees. It works on top of the DataNodeStore and offers an interface which can be used to load, modify and store whole trees as opposed to single nodes. So this is the layer containing the tree algorithms for accessing and resizing trees that are described in Sections 2.9 and 2.10. The reference in Section 5 describes the layout of the block headers used.

ParallelAccessDataTreeStore

The same race condition described before for a blockstore also exists on the tree level. If an application loads the same tree twice and writes both back, then only the last modification survives. Like ParallelAccessBlockStore for blocks, this layer solves the problem for whole trees by returning the same tree object if it is requested twice.

BlobStore

This is a simple adapter layer mapping the blobstore interface, which is used by the application, to the interface offered by the DataTree stores.

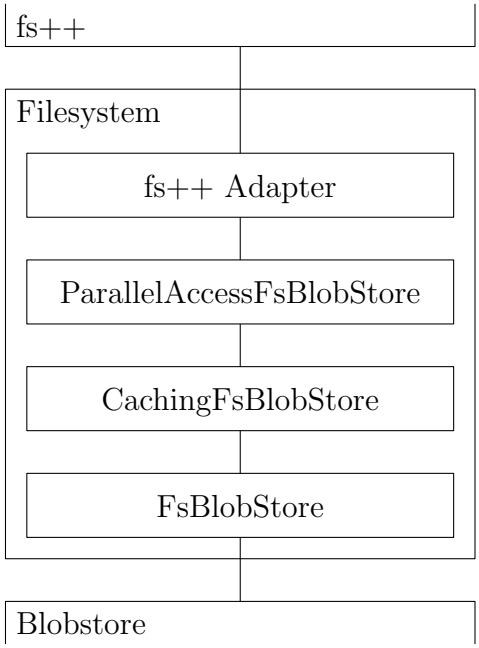


Figure 6.4 Filesystem Architecture. It is divided into sublayers. FsBlobStore offers access to filesystem blobs (i.e. directory blobs, file blobs or symlink blobs). CachingFsBlobStore adds a caching layer and ParallelAccessFsBlobStore takes care of race conditions when accessing the same directory blob in multiple threads. The fs++ Adapter layer implements the fs++ filesystem interface using these filesystem blobs.

6.1.3 Filesystem

This layer implements the filesystem interface offered by fs++ on top of a blobstore. It follows the directory blob approach described in Section 4.6.2 and is split into sublayers as shown in Figure 6.4.

FsBlobStore

This layer works directly on a blobstore, but instead of offering blobs, it offers loading file blob, symlink blob and directory blob objects. The reference in Section 5 describes the layout of the blob headers used.

CachingFsBlobStore

Each time a directory is accessed, the whole directory blob is loaded from the disk into an in-memory directory structure and afterwards written back if it was changed. The CachingFsBlobStore layer takes care that this only happens once if the same directory is accessed multiple times in a short timespan.

ParallelAccessFsBlobStore

The same race condition described before for blockstores and datatreestores also has to be solved on this level. If an application loads the same directory blob twice and writes both back, then only the last modification survives. Like ParallelAccessBlockStore for blocks, this layer solves the problem for directory blobs.

fs++ Adapter

This is a layer implementing the platform independent fs++ filesystem interface. It uses the file blobs, symlink blobs and directory blobs as offered by the layers below to implement filesystem operations.

6.2 Performance Evaluation

This section contains a performance evaluation of CryFS. Since the implementation of CryFS still has potential for performance optimizations, these experiments are not showing the actual performance of a future final version of CryFS. Furthermore, it is generally difficult to evaluate filesystem performance, because there are many different scenarios, e.g. average file size or different access patterns, and filesystems can behave very differently in different scenarios. The experiments are however able to show that—although CryFS is more complex than the other filesystems—it still has good performance and can be used in practice.

6.2.1 Experiment Setup

We tested CryFS 0.8 against EncFS 1.7.4, TrueCrypt 7.1a and VeraCrypt 1.15. CryFS was built with GCC 4.9.2 using optimization level O3. The ciphertexts are stored in an Ext4 filesystem. For an upper bound, we also tested the performance of Ext4 itself without using a cryptographic filesystem on top. The automated benchmark script can be found on github[3].

CryFS has been run using aes-256-gcm. EncFS was also set to aes-256, and they use a custom stream cipher mode which cannot be configured. For TrueCrypt and VeraCrypt, a container with 35 GB size was created, also using the aes-256.

The experiments are run using the benchmarking tool bonnie++ 1.03e[4]. Bonnie++ tests sequential read and write speed, both for bytewise and blockwise reads/writes. It tests the number of random seeks the filesystem is able to perform per second, and how many files the filesystem is able to create, delete, or stat per second. For all these tests, it also reports the average CPU utilization.

The experiments have been run on a machine with Intel (R) Core(TM) i5-2500K CPU @ 3.30GHz QuadCore, 16GB (4x4GB) DDR3-RAM on Ubuntu 15.04, Linux 3.19.0-22 x86_64. Since filesystems can behave very differently on a SSD than on a HDD, we tested both, using a Samsung SpinPoint M8 ST1000LM024 and a Samsung SSD 840 EVO 1TB.

Bonnie is run with the following command line options:

```
-d [test directory]      # Choose HDD/SSD mount location
-x 3                     # Run three times
-n 16:10240:10240:10     # Use 16*1024 files with 10KB each for create/stat/delete test
```

To minimize the influence of cache effects, bonnie++ runs the read/write tests with a test file size that is twice the memory size (i.e. 32GB). Each experiment has been run three times to ensure consistency of the results.

[3] https://github.com/cryfs/benchmark
[4] http://www.coker.com.au/bonnie++/

6.2.2 Read and Write Tests

Table 6.1 shows the HDD performance and Table 6.2 the SSD performance of read/write operations, both bytewise and blockwise. It also shows the results of a *Rewrite* run, which iteratively loads a block from the file, modifies it, and writes it back. Each line shows one run of the experiment, containing read/write speeds and average CPU utilization.

On a SSD, the bytewise read/write tests do not give meaningful results, because they do a syscall for each byte and this is CPU bound at about 100MB/s, even for plain ext4. Also on a HDD, CPU utilization for bytewise operations is high. However, we see that the libFUSE filesystems (EncFS and CryFS) are a bit slower even, probably due to the additional overhead for a libFUSE syscall. A libFUSE read/write syscall first calls into the kernel code, and then back into the filesystem code that is running in userspace. The filesystem operation result then is again routed to the kernel and from there back to the userspace application that issued the syscall. Non-libFUSE filesystems like TrueCrypt, VeraCrypt or plain ext4 have their filesystem implementation in the kernel and do not have to call into a userspace filesystem implementation.

On a HDD, the write performance of CryFS is 20% slower than EncFS while the read performance is about a factor of 2-3 slower, but still fast enough to be used in practice. On a SSD, the write performance of CryFS is about 40% faster than EncFS, while read performance is about a factor of 2 slower. The non-libFUSE filesystems TrueCrypt and VeraCrypt are faster than both. CryFS is fast enough to be used in practice.

6.2.3 Seek, Create, Stat and Delete Tests

Table 6.4 shows the SSD performance of random seeks and of operations that work with directory structure; i.e. creating and deleting files, and reading file attributes (stat). The tests create, stat and delete 16384 files. If the tests created empty files, EncFS is much faster than in the experiments shown here, because it only has to create empty ciphertext files without doing any encryption. But then, this test would not be very close to practical use cases. This is why we configured bonnie++ to create files with a size of 10KB.

Random seeks in CryFS are 20% slower on a HDD and 40% slower on a SSD compared to EncFS. Random create in CryFS is comparable on a HDD and 35% faster than EncFS on a SSD. Random stats are slower by 40% on a HDD and comparable to EncFS on a SSD. Random delete is slower by a factor of 5 on a HDD and 3 on a SSD, but still fast enough to be used in practice.

Creating and deleting files with TrueCrypt and VeraCrypt is strongly CPU bound, while CryFS and EncFS have a better CPU utilization per operation. Interestingly, although seeks are much faster on a SSD, creating and deleting files on a HDD and a SSD have comparable performance.

6.2.4 Conclusion

Some filesystems operations in CryFS compare a bit faster, some a bit slower to EncFS. Most are in a comparable performance range. Deleting files is the exception, being slower by a factor of 3-5. But also the delete operation is fast enough for CryFS to be used in practice.

For doing the same number of operations, CryFS uses a bit less CPU than EncFS and much less CPU than TrueCrypt or VeraCrypt.

	Sequential Output				Sequential Input				Rewrite	
	bytewise		blockwise		bytewise		blockwise		blockwise	
	MB/s	CPU	MB/s	CPU	MB/s	CPU	MB/s	CPU	MB/s	CPU
CryFS	34.5	30%	29.7	4%	30.3	37%	33.6	2%	19.3	4%
	31.7	28%	28.6	3%	27.9	35%	30.4	2%	18.3	3%
	39.7	35%	39.3	4%	46.5	48%	56.1	3%	20.5	4%
EncFS	33.2	35%	50.0	8%	83.9	81%	99.7	3%	32.9	5%
	32.8	36%	51.6	8%	84.5	82%	96.6	3%	32.2	5%
	33.2	36%	49.3	8%	84.6	81%	93.9	4%	31.8	5%
TrueCrypt	69.0	53%	90.6	8%	99.0	77%	113.4	6%	40.0	6%
	90.7	80%	94.4	8%	98.3	76%	112.7	7%	40.7	6%
	89.7	78%	94.8	8%	94.4	74%	112.7	7%	40.3	6%
VeraCrypt	68.3	52%	80.7	7%	86.6	70%	101.5	6%	37.9	5%
	83.6	70%	85.5	7%	86.6	69%	101.6	6%	38.1	5%
	85.8	72%	84.9	7%	86.5	69%	101.4	6%	37.6	5%
Plain Ext4	65.2	60%	100.4	6%	92.1	73%	118.2	5%	41.9	4%
	70.4	53%	99.0	6%	95.1	74%	114.9	5%	39.8	4%
	70.0	53%	96.5	6%	89.7	71%	111.7	5%	39.0	4%

Table 6.1 Performance experiments for read/write operations on a HDD. Each line is one experiment run. The *Rewrite* test iteratively reads, modifies and writes back a block of data. Write performance of CryFS is 20% slower than EncFS and read performance is about a factor of 2-3 slower, but still fast enough to be used in practice. TrueCrypt and VeraCrypt need some warmup time and then are for bytewise reads/writes even faster than native Ext4. This is surprising, but possible, because they manage only one fixed-size container file in the Ext4 partition. They do not run Ext4 operations to access and modify many ciphertext files like CryFS or EncFS do.

	Sequential Output				Sequential Input				Rewrite	
	bytewise		blockwise		bytewise		blockwise		blockwise	
	MB/s	CPU	MB/s	CPU	MB/s	CPU	MB/s	CPU	MB/s	CPU
CryFS	49.3	49%	75.8	9%	94.6	95%	125.9	5%	33.1	6%
	47.6	49%	81.5	10%	93.2	97%	123.8	5%	34.9	6%
	45.0	46%	80.6	10%	95.9	97%	120.0	5%	34.5	6%
EncFS	35.4	41%	57.3	10%	95.4	99%	215.6	5%	41.5	7%
	34.8	42%	57.3	10%	95.7	99%	215.9	5%	41.6	7%
	34.7	41%	57.3	10%	95.9	99%	220.2	5%	42.0	7%
TrueCrypt	108.6	96%	358.0	24%	104.0	98%	587.5	15%	198.2	16%
	102.5	98%	273.5	20%	108.6	99%	592.1	15%	195.6	15%
	101.6	97%	264.3	19%	107.3	99%	591.1	15%	202.2	16%
VeraCrypt	110.1	97%	362.8	26%	104.1	98%	591.2	16%	207.4	17%
	103.0	98%	350.7	25%	103.7	99%	596.2	15%	207.0	17%
	102.4	98%	341.7	25%	104.7	98%	594.7	16%	205.6	16%
Plain Ext4	118.5	94%	361.3	21%	121.4	98%	652.4	27%	215.7	19%
	109.0	92%	402.2	24%	122.1	99%	648.8	27%	214.8	19%
	113.0	94%	391.6	22%	121.5	99%	653.0	27%	214.6	19%

Table 6.2 Performance experiments for read/write operations on a SSD. Each line is one experiment run. The *Rewrite* test iteratively reads, modifies and writes back a block of data. Performance for bytewise reads and writes is CPU bound at 100 MB/s, even on plain ext4. CryFS fast enough to be used in practice. It is slower in reading and faster in writing than EncFS. Filesystems not using libFUSE (TrueCrypt, VeraCrypt) are faster.

	Random Seeks		Random Create		Random Stat		Random Delete	
	/s	CPU	/s	CPU	/s	CPU	/s	CPU
CryFS	72.7	0%	3603	11%	5738	13%	3633	6%
	70.0	0%	4087	11%	5687	12%	3781	6%
	70.9	0%	3963	11%	5778	13%	3546	6%
EncFS	92.8	0%	3836	15%	9419	20%	17038	22%
	90.5	0%	3860	15%	9378	20%	17432	21%
	89.1	0%	3928	15%	9195	20%	17303	21%
TrueCrypt	104.5	0%	4231	99%	+	+	9913	99%
	102	0%	4242	99%	+	+	9887	100%
	99.6	0%	4222	99%	+	+	9735	99%
VeraCrypt	105.9	0%	4234	99%	+	+	9947	100%
	104.5	0%	4201	99%	+	+	9871	100%
	103.7	0%	1920	46%	+	+	9941	100%
Plain Ext4	127.4	0%	+	+	+	+	+	+
	143.7	0%	+	+	+	+	+	+
	130.4	0%	+	+	+	+	+	+

+) Test finished too fast to give meaningful results.

Table 6.3 Performance experiments for random seeks and file create/stat/delete on a HDD. Each line is one experiment run. CryFS performance for seek is 20% slower, create is comparable to EncFS, stat is 40% slower and delete is slower by a factor of 5. However, all operations are fast enough for CryFS to be used in practice. Performance of TrueCrypt and VeraCrypt for create/delete is CPU bound while EncFS is more CPU efficient and CryFS even a bit better.

	Random Seeks		Random Create		Random Stat		Random Delete	
	/s	CPU	/s	CPU	/s	CPU	/s	CPU
CryFS	1286	2%	3981	11%	5776	13%	3700	6%
	1997	2%	4028	11%	5749	13%	3791	6%
	1754	2%	4054	11%	5788	13%	3575	6%
EncFS	2757	5%	3074	15%	4864	20%	7749	21%
	2741	5%	1974	15%	6142	20%	11077	21%
	2688	5%	3332	15%	6139	20%	12232	20%
TrueCrypt	5421	6%	4237	99%	+	+	9895	100%
	5418	5%	4251	99%	+	+	6546	67%
	5274	6%	4214	99%	+	+	9850	100%
VeraCrypt	5484	7%	4235	99%	+	+	9856	100%
	5273	6%	4250	99%	+	+	5345	58%
	5397	6%	4223	99%	+	+	9809	100%
Plain Ext4	+	+	+	+	+	+	+	+
	+	+	+	+	+	+	+	+
	+	+	+	+	+	+	+	+

+) Test finished too fast to give meaningful results.

Table 6.4 Performance experiments for random seeks and file create/stat/delete on a SSD. Each line is one experiment run. Seeks in CryFS are slower than EncFS by 40%, creates are faster by 35%, stat is comparable and delete is slower by a factor of 3. All operations are fast enough for CryFS to be used in practice. Performance of TrueCrypt and VeraCrypt for create/delete is CPU bound while EncFS is more CPU efficient and CryFS even a bit better.

7. Security Analysis

In this section, we prove the security properties of CryFS. We show that CryFS inherits the indistinguishability properties from the symmetric encryption scheme used to encrypt the blocks. If this encryption scheme is IND-atk secure, then CryFS is IND-atk secure ($\mathsf{atk} \in \{\mathsf{CPA}, \mathsf{CCA1}\}$). We also introduce an assumption and show that under this assumption, CryFS also inherits IND-CCA2 and INT-CTXT security. If the blocks are encrypted with a scheme that is only INT-PTXT but not INT-CTXT secure, then we need a stronger assumption to show that CryFS inherits INT-PTXT security.

7.1 Model

This section provides a formal definition of CryFS. In Definition 7.1, we define a generic symmetric encryption scheme and specialize it in Definition 7.2 to encrypt blocks; i.e. tuples of a block ID and data. We extend this to a scheme encrypting sets of blocks; see Definition 7.3. Then, in Definition 7.4, we introduce a scheme that encrypts a plaintext p from an arbitrary plaintext space by applying an arbitrary function f mapping p to a set of blocks and then encrypting the set of blocks. This is then used to define CryFS, with f being the representation of the filesystem in plaintext blocks; see Definition 7.5. In later sections, we use this chain of definitions to show for each definition that it is secure if the previous one is secure. Using this, we show that CryFS is secure, if the underlying symmetric encryption scheme to encrypt the blocks is secure. All the schemes we define are specializations of the symmetric encryption scheme in Definition 7.1.

Definition 7.1 (symmetric encryption scheme)
Let S be a plaintext space and T be a ciphertext space. A symmetric encryption scheme C is a tuple $\mathsf{C} = (\mathsf{Enc}_k^{\mathsf{C}}, \mathsf{Dec}_k^{\mathsf{C}})$ of two functions with

$$\mathsf{Enc}_k^{\mathsf{C}} : S \to T \qquad \mathsf{Dec}_k^{\mathsf{C}} : T \to (\{\bot\} \cup S)$$

The encryption function $\mathsf{Enc}_k^{\mathsf{C}}$ can be indeterministic, but each evaluation has to fulfill the following condition.

$$\forall k \forall s \in S : \mathsf{Dec}_k^{\mathsf{C}}(\mathsf{Enc}_k^{\mathsf{C}}(s)) = s$$

Decryption can fail and return $\bot$, for example if integrity is violated. The parameter k represents the encryption key.

In the following Definition 7.2, we define the specialization of a symmetric encryption scheme for blocks. A block consists of a block ID and some data. For integrity reasons, it additionally concatenates the block ID with the data before encrypting it. Let ρ be the length of a block ID and $\mathbb{K} = \{0,1\}^\rho$ be the set of all possible block IDs. Let $\mathbb{B}^L := \mathbb{K} \times \{0,1\}^L$ be the set of all possible data blocks containing exactly L bytes and let $\mathbb{B}^* := \bigcup_L \mathbb{B}^L$ be the set of all possible data blocks. Let $\mathsf{id} : \mathbb{B}^* \to \mathbb{K}$ and $\mathsf{data} : \mathbb{B}^* \to \{0,1\}^*$ be functions returning the ID respectively the data of a block. For any $L \in \mathbb{N}$, let $\mathsf{g} : \mathbb{B}^L \to \{0,1\}^{\rho+L}$ be a bijective function concatenating ID and data of a block. The function g is needed, because the block encryption scheme in Definition 7.2 concatenates the block ID with the block data before encrypting it. The reason for this is to inherit integrity properties of the encryption scheme for the block ID. It prevents attackers from changing the ID of a block or reintroducing old blocks to the system with a new ID.

Definition 7.2 (block encryption scheme)
For a symmetric encryption scheme $\mathsf{C} = (\mathsf{Enc}_k^{\mathsf{C}}, \mathsf{Dec}_k^{\mathsf{C}})$ *that operates on* $S = T = \{0,1\}^*$, *the block encryption scheme* $\mathsf{C}' = (\mathsf{Enc}_k^{\mathsf{C}'}, \mathsf{Dec}_k^{\mathsf{C}'})$ *is defined as follows:*

$$\mathsf{Enc}_k^{\mathsf{C}'} : \mathbb{B}^L \to \mathbb{B}^* \qquad \mathsf{Dec}_k^{\mathsf{C}'} : \mathbb{B}^* \to (\{\bot\} \cup \mathbb{B}^L)$$

$$\mathsf{Enc}_k^{\mathsf{C}'}((i,m)) := (i, \mathsf{Enc}_k^{\mathsf{C}}(\mathsf{g}(i,m))$$

$$\mathsf{Dec}_k^{\mathsf{C}'}((i,c)) := \begin{cases} \bot & \text{if } \mathsf{Dec}_k^{\mathsf{C}}(c) = \bot \\ \bot & \text{if } \mathsf{id}(\mathsf{g}^{-1}(\mathsf{Dec}_k^{\mathsf{C}}(c))) \neq i \\ \mathsf{g}^{-1}(\mathsf{Dec}_k^{\mathsf{C}}(c)) & \text{otherwise} \end{cases}$$

The following Definition 7.3 defines an encryption scheme encrypting a set of blocks. The notation $\mathcal{P}(M)$ is used to describe the power set of M, i.e. the set of all subsets of M. So $\mathcal{P}(\mathbb{B}^L)$ is the set of all possible sets of blocks of length L and $\mathcal{P}(\mathbb{B}^*)$ is the set of all possible sets of blocks. By using sets and not vectors of blocks, it is expressed that the blocks do not need to be ordered.

Definition 7.3 (multi-block encryption scheme)
Let $\mathsf{C}' = (\mathsf{Enc}_k^{\mathsf{C}'}, \mathsf{Dec}_k^{\mathsf{C}'})$ *be a block encryption scheme as defined in Definition 7.2. The multi-block encryption scheme* $\mathsf{C}^* = (\mathsf{Enc}_k^{\mathsf{C}^*}, \mathsf{Dec}_k^{\mathsf{C}^*})$ *is then defined as follows:*

$$\mathsf{Enc}_k^{\mathsf{C}^*} : \mathcal{P}(\mathbb{B}^L) \to \mathcal{P}(\mathbb{B}^*) \qquad \mathsf{Dec}_k^{\mathsf{C}^*} : \mathcal{P}(\mathbb{B}^*) \to (\{\bot\} \cup \mathcal{P}(\mathbb{B}^L))$$

$$\mathsf{Enc}_k^{\mathsf{C}^*}(\{m_1, \ldots, m_r\}) := \{\mathsf{Enc}_k^{\mathsf{C}'}(m_1), \ldots, \mathsf{Enc}_k^{\mathsf{C}'}(m_r)\}$$

$$\mathsf{Dec}_k^{\mathsf{C}^*}(\{c_1, \ldots, c_r\}) := \begin{cases} \bot & \text{if } \exists c_i : \mathsf{Dec}_k^{\mathsf{C}'}(c_i) = \bot \\ \{\mathsf{Dec}_k^{\mathsf{C}'}(c_1), \ldots, \mathsf{Dec}_k^{\mathsf{C}'}(c_r)\} & \text{otherwise} \end{cases}$$

Having defined a scheme to encrypt sets of blocks, we now can encrypt filesystem states by mapping them to sets of blocks. This is done by the following Definition 7.4.

Definition 7.4 (f-multi-block encryption scheme)
Let C^* be a multi-block encryption scheme as defined in Definition 7.3. Let $\mathbb{M}$ be any plaintext space and for any $L \in \mathbb{N}$, let $\mathsf{f} : \mathbb{M} \to \mathcal{P}(\mathbb{B}^L)$ be an injective function mapping elements from $\mathbb{M}$ to sets of blocks. The f-multi-block encryption scheme $\mathsf{C}^\mathsf{f} = (\mathsf{Enc}_k^{\mathsf{C}^\mathsf{f}}, \mathsf{Dec}_k^{\mathsf{C}^\mathsf{f}})$ is then defined as follows:

$$\mathsf{Enc}_k^{\mathsf{C}^\mathsf{f}} : \mathbb{M} \to \mathcal{P}(\mathbb{B}^*) \qquad \mathsf{Dec}_k^{\mathsf{C}^\mathsf{f}} : \mathcal{P}(\mathbb{B}^*) \to (\{\bot\} \cup \mathbb{M})$$

$$\mathsf{Enc}_k^{\mathsf{C}^\mathsf{f}}(m) := \mathsf{Enc}_k^{\mathsf{C}^*}(\mathsf{f}(m))$$

$$\mathsf{Dec}_k^{\mathsf{C}^\mathsf{f}}(c) := \begin{cases} \bot & \text{if } \mathsf{Dec}_k^{\mathsf{C}^*}(c) = \bot \\ \mathsf{f}^{-1}(\mathsf{Dec}_k^{\mathsf{C}^*}(c)) & \text{otherwise} \end{cases}$$

In the following, we specialize Definition 7.4 for CryFS. Let now $\mathbb{M}$ be the set of all possible filesystem states. For any $L \in \mathbb{N}$, let f be an injective function mapping a filesystem state to an arbitrary number of L-sized blocks.

$$\mathsf{f} : \mathbb{M} \to \mathcal{P}(\mathbb{B}^L)$$

So f can be any way to represent a filesystem state using same size blocks. It is important to note that a filesystem state includes all relevant information. It does not only include the file contents, but also directory structure and file metadata. The CryFS representation is not modeled in more detail, because this is sufficient to proof the security of CryFS.

Definition 7.5 (CryFS)
For a symmetric encryption scheme $\mathsf{C} = (\mathsf{Enc}_k^{\mathsf{C}}, \mathsf{Dec}_k^{\mathsf{C}})$, let C^f be the corresponding f-multi-block encryption scheme as defined in Definition 7.4. The function f is defined as above and maps filesystem states to sets of blocks. Then, the $\mathsf{CryFS}^\mathsf{C}$ encryption scheme is defined as $\mathsf{CryFS}^\mathsf{C} := \mathsf{C}^\mathsf{f}$.

7.2 Attacker Restrictions

This section introduces some attacker restrictions and assumptions that are needed as a basis for the security proofs. We also explain what the attacker restrictions mean in practice and why the assumptions are justified.

7.2.1 Confidentiality

Since an attacker can easily distinguish two filesystems of different sizes or two filesystems that use different block IDs, we need some restrictions on attackers. The following Definition 7.6 explains restrictions put on IND-atk attackers.

Definition 7.6 (Attacker restrictions)

- *In the* IND-atk *game against a block encryption scheme* C'*, the attacker is restricted to choosing two plaintext blocks with the same ID.*
- *In the* IND-atk *game against a multi-block encryption scheme* C^**, the attacker is restricted to choosing two plaintext sets with the same number of blocks and the same block IDs.*
- *In the* IND-atk *game against a f-multi-block encryption scheme* C^f*, the attacker is restricted to choosing two plaintexts* m_1, m_2 *where* $f(m_1), f(m_2)$ *have the same number of blocks and the same block IDs.*
- *In the* IND-atk *game against* CryFS*, the attacker is restricted to choosing two filesystem states* m_1, m_2 *where* $f(m_1), f(m_2)$ *have the same number of blocks and the same block IDs.*

The restrictions against $\mathsf{C}', \mathsf{C}^*$ and C^f are only important for intermediate results. We show in Section 7.3 that if there is no IND-atk attacker against the symmetric encryption scheme C, then there is no IND-atk attacker against the block encryption scheme C' with the restriction from Definition 7.6. If there is no IND-atk attacker against C' with this restriction, then there is no IND-atk attacker against the multi-block encryption scheme C^* with this restriction. If there is no IND-atk attacker against C^* with this restriction, then there is no IND-atk attacker against the f-multi-block encryption scheme C^f with this restriction. And finally, if there is no IND-atk attacker against C^f with this restriction, then there is no IND-atk attacker against CryFS with this restriction. So the only restriction in the final result is the restriction against CryFS.

The restriction to output two plaintext filesystem states that have the same number of blocks effectively means that CryFS does not hide the filesystem size. The restriction that the two filesystem states have to have the same block IDs means that CryFS does not hide the block IDs. Since CryFS chooses the block IDs randomly and independently from the actual filesystem state, this is not a problem.

7.2.2 Integrity

CryFS implements integrity as defined in Section 4.4.1. In summary: an authenticated block cipher prevents an attacker from manipulating blocks, storing the block ID in the (authenticated) block header prevents an attacker from replacing a block with a different block, version counters in the block header prevent an attacker from rolling back blocks, and an integrity-checked list of deleted blocks prevents an attacker from deleting or undeleting blocks.

This implementation however needs local state in the client. To keep formalization of CryFS simple, we do not use a stateful formalization but instead make the following Assumption 1. This assumption is used in Lemma 7.15 when showing that INT-PTXT security of multi-block encryption follows from INT-PTXT security of the underlying block encryption.

Assumption 1 (for plaintext integrity)
An attacker against a multi-block encryption scheme cannot output or query a decryption oracle for $\{c_1, \ldots, c_n\}$ *which decrypts to* $\{p_1, \ldots, p_n\}$ *if there was an oracle input* $\{p'_1, \ldots, p'_m\}$ *with at least one* $p'_i = p_j$*.*

Intuitively, this assumption prevents an attacker from recombining blocks from two different filesystem states at different times. Because of the integrity measures described, we believe this to be true. The only way for an attacker to get valid encrypted blocks is to take them from past states of the filesystem. If a filesystem state was $p' := \{p'_1, \ldots, p'_m\}$ in the past and the attacker fakes a ciphertexts that decrypts to $p := \{p_1, \ldots, p_n\}$ with at least one $p'_i = p_j$, then it could only have done that by taking the $p_j \notin p'$ from other, past filesystem states. However, because of the integrity measures described above, that means that these blocks have either invalid version counters, invalid IDs in their headers, or are invalid due to the integrity-checked list of deleted blocks. That is, an attacker is unable to fake a valid $\{p_1, \ldots, p_n\}$ and we believe the assumption to be true.

For showing that INT-CTXT security of multi-block encryption follows from INT-CTXT security of the underlying block encryption, the following weaker Assumption 2 is enough. As shown in Lemma 7.7, Assumption 1 implies Assumption 2. Furthermore, Lemma 7.15 proves a result that is true under Assumption 1 while Lemma 7.14 shows that Assumption 2 would not be enough for this proof. That is, Assumption 2 is strictly weaker than Assumption 1.

Assumption 2 (for ciphertext integrity)
An attacker against a multi-block encryption scheme cannot output or query a decryption oracle for $\{c_1, \ldots, c_n\}$ if there was an oracle output $\{c'_1, \ldots, c'_m\}$ with at least one $c'_i = c_j$.

Lemma 7.7
If Assumption 1 is true, then Assumption 2 is true as well.

Proof Let Assumption 1 be true. Let there be an integrity attacker against a multi-block encryption scheme who outputs or queries a decryption oracle for $\{c_1, \ldots, c_n\}$ which decrypts to $\{p_1, \ldots, p_n\}$. Assume there was an oracle output $\{c'_1, \ldots, c'_m\}$ which decrypts to $\{p'_1, \ldots, p'_m\}$ with at least one $c'_i = c_j$ and lead this to a contradiction. Because $\{c'_1, \ldots, c'_m\}$ was oracle output, $\{p'_1, \ldots, p'_m\}$ was oracle input. Because $c'_i = c_j$, we also know $p'_i = p_j$. This is a contradiction with Assumption 1. ∎

It might be possible to imply these assumptions from weaker assumptions that only forbid recombining two known plaintexts, i.e. recombining two past oracle inputs. This is left to future work.

7.3 Confidentiality

In this section, we show that CryFS fulfills the confidentiality goals. Theorem 1 is the core of this section. It shows that when the blocks of CryFS are encrypted with an IND-atk secure encryption scheme C, then CryFSC itself is IND-atk secure. To prove this theorem, we build a chain of lemmata showing the following:

$$\mathsf{C}\text{ IND-atk} \overset{7.8}{\Longrightarrow} \mathsf{C}'\text{ IND-atk} \overset{7.9/7.10}{\Longrightarrow} \mathsf{C}^*\text{ IND-atk} \overset{7.11}{\Longrightarrow} \mathsf{C}^{\mathsf{f}}\text{ IND-atk} \overset{\text{Th.1}}{\Longrightarrow} \mathsf{CryFS}^{\mathsf{C}}\text{ IND-atk}$$

Throughout this section, if not otherwise defined, it is $\mathsf{atk} \in \{\mathsf{CPA}, \mathsf{CCA1}, \mathsf{CCA2}\}$. For all attackers, the restrictions from Definition 7.6 apply.

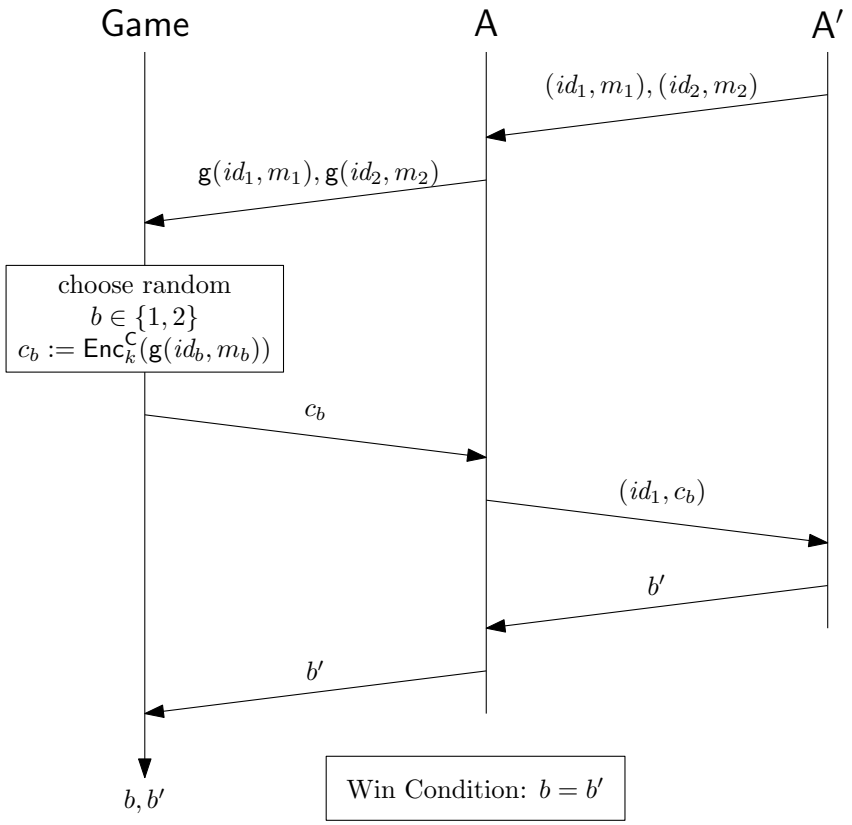


Figure 7.1 Proof for block indistinguishability. An IND-atk attacker A against C is built by using an IND-atk attacker A′ against C′. As defined for block encryption schemes, A passes through the concatenated block ID and data to be encrypted by the game. Oracle queries are not shown.

Lemma 7.8 (C is IND-atk ⇒ C′ is IND-atk)
Let $\mathsf{C} = (\mathsf{Enc}_k^{\mathsf{C}}, \mathsf{Dec}_k^{\mathsf{C}})$ *be a symmetric encryption scheme which is* IND-atk *secure. Let* $\mathsf{C}' = (\mathsf{Enc}_k^{\mathsf{C}'}, \mathsf{Dec}_k^{\mathsf{C}'})$ *be the corresponding block encryption scheme. Then,* C′ *is* IND-atk *secure.*

Proof Let A′ be an attacker who wins the IND-atk game against C′. Then, an attacker A as shown in Figure 7.1 wins the IND-atk game against C.

A′ chooses two plaintexts (id_1, m_1), (id_2, m_2) and sends them to A. A then forwards $\mathsf{g}(id_1, m_1), \mathsf{g}(id_2, m_2)$ to the game and gets the challenge c. It forwards (id_1, c) to A′. This challenge is well formed, because Definition 7.6 restricts A′ to choosing two plaintext blocks with the same block ID; i.e. $id_1 = id_2$. A′ guesses b', passes it to A, which forwards it to the game. A wins exactly if A′ wins. ∎

Lemma 7.9 (C is IND-CPA/CCA1 ⇒ C* is IND-CPA/CCA1)
Let $\mathsf{C} = (\mathsf{Enc}_k^{\mathsf{C}}, \mathsf{Dec}_k^{\mathsf{C}})$ *be a symmetric encryption scheme which is* IND-atk *secure (*$\mathsf{atk} \in \{\mathsf{CPA}, \mathsf{CCA1}\}$*). Let* $\mathsf{C}^* = (\mathsf{Enc}_k^{\mathsf{C}^*}, \mathsf{Dec}_k^{\mathsf{C}^*})$ *be the corresponding multi-block encryption scheme. Then,* C* *is* IND-atk *secure.*

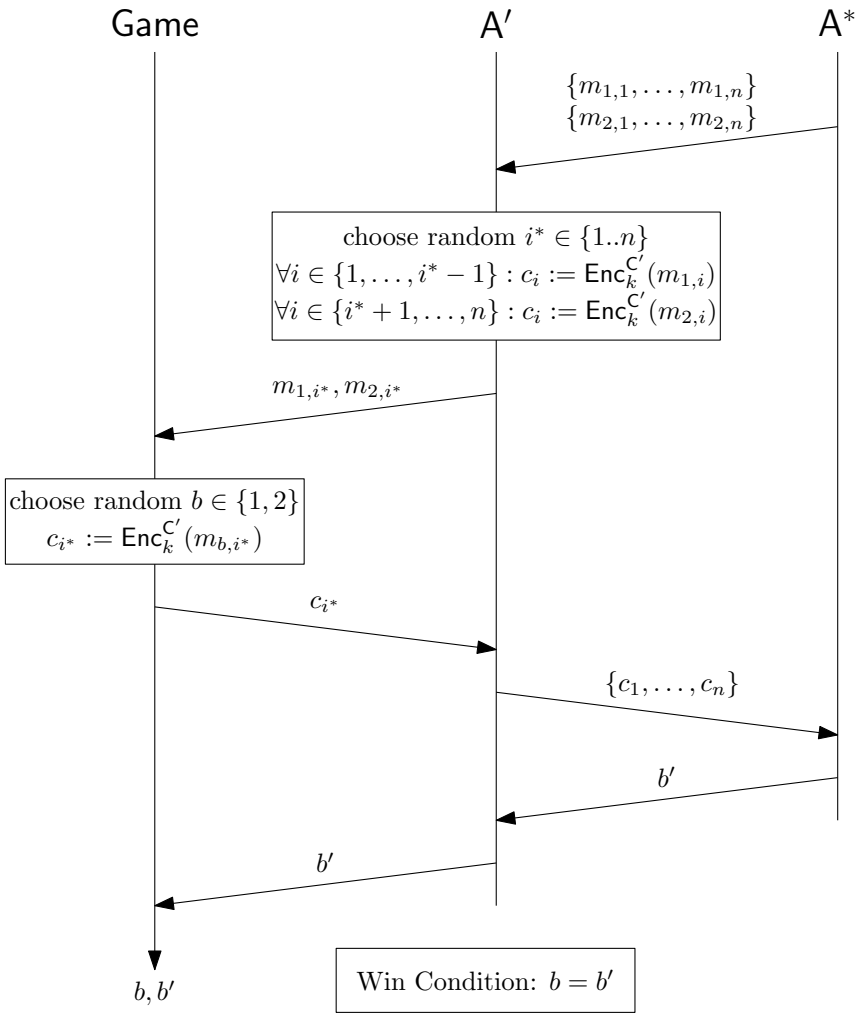


Figure 7.2 Proof for multi-block indistinguishability. An IND-atk attacker A' against C' is built by using an IND-atk attacker A^* against C^*. Out of the two plaintext sets chosen by A^*, one random index is chosen by A' and forwarded to the game. For the other indices, the encryption oracle is used. Oracle queries are not shown.

Proof Let $\mathsf{C}' = (\mathsf{Enc}^{\mathsf{C}'}, \mathsf{Dec}^{\mathsf{C}'})$ be the corresponding block encryption scheme. With Lemma 7.8, we know that C' is IND-atk secure. That is, it is enough to reduce an IND-atk attacker against C^* to an IND-atk attacker against C'.

Let A^* be an attacker who wins the IND-atk game against C^*. We build an attacker A' who wins the IND-atk game against C' as described in Figure 7.2.

The oracles used by A^* before it gets the challenge can be directly build using the corresponding oracles for A'. The attacker A' plays the game as follows: A' asks A^* for $\widetilde{m_1} = \{m_{1,1}, \ldots, m_{1,n}\}, \widetilde{m_2} = \{m_{2,1}, \ldots, m_{2,n}\}$. According to the attacker restriction in Definition 7.6, both sets have the same number of entries and the same block IDs. The sets are unordered, so we can w.l.o.g. choose the indices so that for all i: $\mathsf{id}(m_{1,i}) = \mathsf{id}(m_{2,i})$. Then, A' chooses randomly $i^* \in \{1..n\}$ and builds the ciphertext $\{c_1, \ldots, c_n\}$ by encrypting $i^* - 1$ plaintexts from $\widetilde{m_1}$ and $n - i^*$ from $\widetilde{m_2}$ using its oracle. Since $\widetilde{m_1}$ and $\widetilde{m_2}$ are

unordered sets, it chooses the plaintexts in an arbitrary fashion, w.l.o.g. $m_{1,1} \dots m_{1,i^*-1}$ and $m_{2,i^*+1} \dots m_{2,n}$. Additionally, it encrypts one ciphertext not using its oracle but by sending w.l.o.g. m_{1,i^*} and m_{2,i^*} to the game, which encrypts one of them according to its choice bit b. This is a well formed IND-atk game against C', because it fulfills the attacker restriction $\mathsf{id}(m_{1,i^*}) = \mathsf{id}(m_{2,i^*})$ in Definition 7.6.

$$\forall i \in \{1, \dots, i^* - 1\} : c_i := \mathsf{Enc}_k^{\mathsf{C}'}(m_{1,i})$$
$$c_{i^*} := \mathsf{Enc}_k^{\mathsf{C}'}(m_{b,i^*})$$
$$\forall i \in \{i^* + 1, \dots, n\} : c_i := \mathsf{Enc}_k^{\mathsf{C}'}(m_{2,i})$$

Then, A' sends $\{c_1, \dots, c_n\}$ as a challenge to A^*. Attacker A^* guesses b', which A' forwards as its guess to the game.

Now we calculate the win probability for A'. Let $p_{b'}^{i^*}$ be the probability that A^* outputs b' given that i^* messages have been encrypted from $\widetilde{m_1}$ and $n - i^*$ messages from $\widetilde{m_2}$. Now for $i^* \notin \{0, n\}$, A^* gets a malformed challenge. However, the following calculations show that this does not matter. When $b = 1$, we know that at least one plaintext was encrypted from $\widetilde{m_1}$ and analogously for $\widetilde{m_2}$ when $b = 2$, so

$$\mathsf{Pr}[\mathsf{A}^* \text{ outputs } 1 | b = 1] = \frac{1}{n-1} \sum_{i^*=2}^{n} p_1^{i^*}$$

$$\mathsf{Pr}[\mathsf{A}^* \text{ outputs } 2 | b = 2] = \frac{1}{n-1} \sum_{i^*=1}^{n-1} p_2^{i^*}$$

In case $i^* = 0$ or $i^* = n$, A^* is given a correctly formed challenge, because it is a valid encryption of $\widetilde{m_0}$ respective $\widetilde{m_1}$. So we have

$$\mathsf{Pr}[\mathsf{A}^* \text{ wins} | \mathsf{A}^* \text{ is given a correctly formed challenge}] = \frac{1}{2}(p_2^0 + p_1^n)$$

Using this, we can calculate the win probability for A' as follows:

$$\begin{aligned}
\mathsf{Pr}[\mathsf{A}' \text{ wins}] &= \frac{1}{2}(\mathsf{Pr}[\mathsf{A}^* \text{ outputs } 2 | b = 2] \cdot \mathsf{Pr}[\mathsf{A}^* \text{ outputs } 1 | b = 1]) \\
&= \frac{1}{2n}(\sum_{i^*=0}^{n-1} p_2^{i^*} + \sum_{i^*=1}^{n} p_1^{i^*}) \\
&= \frac{1}{2n}(p_2^0 + \sum_{i^*=1}^{n-1} (p_2^{i^*} + p_1^{i^*}) + p_1^n) \\
&= \frac{1}{2n}(p_2^0 + \sum_{i^*=1}^{n-1} (1) + p_1^n) \\
&= \frac{1}{2n}(p_2^0 + (n-1) + p_1^n) \\
&= \frac{n-1}{2n} + \frac{1}{2n}(p_2^0 + p_1^n) \\
&= \frac{n-1}{2n} + \frac{1}{n}\mathsf{Pr}[\mathsf{A}^* \text{ wins} | \mathsf{A}^* \text{ is given a correctly formed challenge}] \\
&= \frac{1}{2} - \frac{1}{2n} + \frac{1}{n}\mathsf{Pr}[\mathsf{A}^* \text{ wins} | \mathsf{A}^* \text{ is given a correctly formed challenge}]
\end{aligned}$$

$$\begin{aligned}\mathsf{Adv}_{\mathrm{IND}}(\mathsf{A}') &= \mathsf{Pr}[\mathsf{A}' \text{ wins}] - \frac{1}{2} \\ &= \frac{1}{n}\mathsf{Pr}[\mathsf{A}^* \text{ wins} | \mathsf{A}^* \text{ is given a correctly formed challenge}] - \frac{1}{2n} \\ &= \frac{1}{n}(\mathsf{Pr}[\mathsf{A}^* \text{ wins} | \mathsf{A}^* \text{ is given a correctly formed challenge}] - \frac{1}{2}) \\ &= \frac{1}{n}\mathsf{Adv}_{\mathrm{IND}}(\mathsf{A}^*)\end{aligned}$$

Because $\mathsf{Adv}_{\mathrm{IND}}(\mathsf{A}^*)$ is non-negligible, $\mathsf{Adv}_{\mathrm{IND}}(\mathsf{A})$ is non-negligible as well. ∎

This proof is similar to a proof of multi-message security in the book of Katz and Lindell [KL08], but adapted to block indistinguishability.

To achieve this result for IND-CCA2 security, Assumption 2 is needed. The following Lemma 7.10 shows this result.

Lemma 7.10 (Assumption 2 and C is IND-CCA2 ⇒ C* is IND-CCA2)
Let $\mathsf{C} = (\mathsf{Enc}_k^{\mathsf{C}}, \mathsf{Dec}_k^{\mathsf{C}})$ *be a symmetric encryption scheme which is* IND-CCA2 *secure. Let* $\mathsf{C}^* = (\mathsf{Enc}_k^{\mathsf{C}^*}, \mathsf{Dec}_k^{\mathsf{C}^*})$ *be the corresponding multi-block encryption scheme. If Assumption 2 is true, then* C^* *is* IND-CCA2 *secure.*

Proof We build the attacker A' from an attacker A^* exactly as shown in the proof for Lemma 7.9 and in Figure 7.2. In the CCA2 game, A^* can use a decryption oracle even after it got the challenge $\{c_1, \ldots, c_n\}$. We build that oracle as follows: Assumption 2 implies that the oracle query $\{q_1, \ldots, q_n\}$ does not contain any q_i with $q_i = c_{i^*}$. So we can decrypt the q_i individually with the decryption oracle of A' and combine them again afterwards. ∎

Lemma 7.11 (C^* is IND-atk ⇒ C^{f} is IND-atk)
Let $\mathsf{C}^* = (\mathsf{Enc}_k^{\mathsf{C}^*}, \mathsf{Dec}_k^{\mathsf{C}^*})$ *be a multi-block encryption scheme which is* IND-atk *secure. Let* C^{f} *be a corresponding f-multi-block encryption scheme. For simplicity of notation, let* $\mathsf{f}^{-1}(\perp) := \perp$. *Then,* C^{f} *is* IND-atk *secure.*

Proof Let A^{f} be an attacker who wins the IND-atk game against C^{f}. Then, an attacker A^* as described in Figure 7.3 wins the IND-atk game against C^*. The oracles for A^{f} can be directly build by applying f or f^{-1} to the corresponding oracle of A^*. A^* asks A^{f} for m_1, m_2 and gives $m_1' := \mathsf{f}(m_1), m_2' := \mathsf{f}(m_1)$ to the game. According to Definition 7.6, m_1' and m_2' have the same block IDs. That is, the plaintexts are valid plaintexts and fulfill the attacker restrictions for the IND-atk game against C^* in Definition 7.6. The game chooses $b \in \{1, 2\}$ and returns $c_b' := \mathsf{Enc}_k^{\mathsf{C}^*}(m_b') = \mathsf{Enc}_k^{\mathsf{C}^{\mathsf{f}}}(m_b)$ to A^*. A^* forwards this to A^{f}. A^{f} sends b' to A^* which A^* forwards to the game. A^* wins against C^* if A^{f} wins against C^{f}. ∎

Theorem 1 (CryFS Confidentiality)
If the blocks are encrypted using an IND-atk *(*atk $\in \{\mathsf{CPA}, \mathsf{CCA1}\}$*) secure symmetric encryption scheme* C*, then* $\mathsf{CryFS}^{\mathsf{C}}$ *is* IND-atk *secure.*

If Assumption 2 is true, then the same is true for IND-CCA2 *security.*

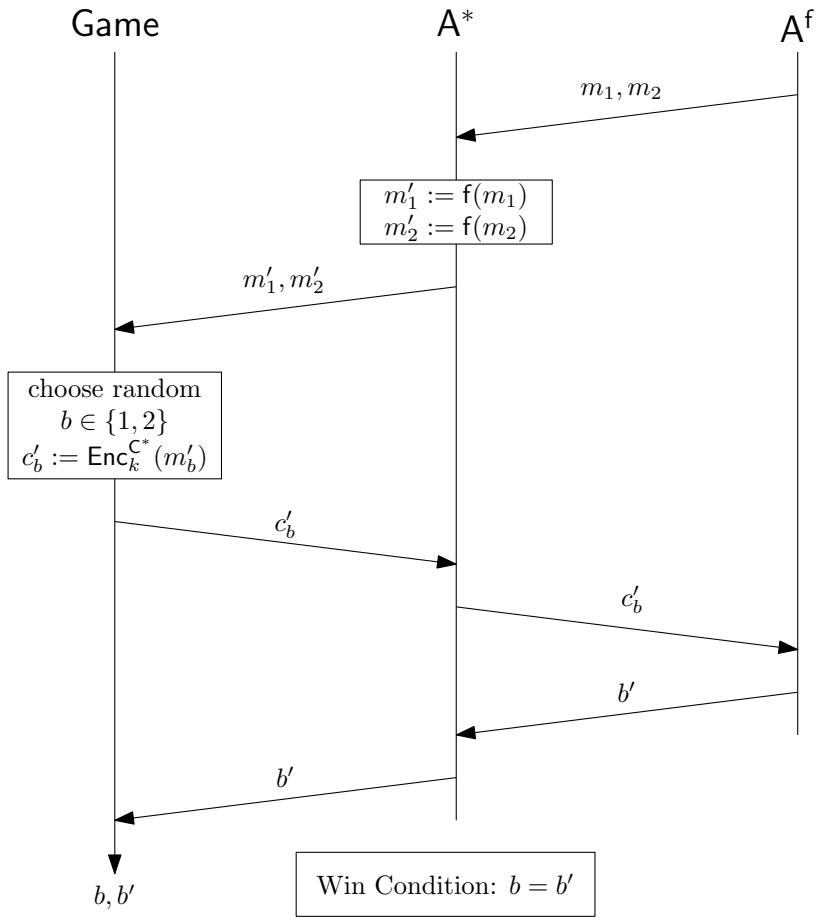


Figure 7.3 Proof for indistinguishability under a function f. An IND-atk attacker A^* against C^* is built by using an IND-atk attacker A^f against C^f. Before passing on the chosen plaintexts, it applies f to them. Oracle queries are not shown.

Proof Let $\mathsf{C} = (\mathsf{Enc}^\mathsf{C}_k, \mathsf{Dec}^\mathsf{C}_k)$ be an IND-atk secure symmetric encryption scheme. Lemma 7.9 implies that the corresponding multi-block encryption scheme $\mathsf{C}^* = (\mathsf{Enc}^{\mathsf{C}^*}_k, \mathsf{Dec}^{\mathsf{C}^*}_k)$ is IND-atk secure. An attacker against CryFS is restricted to choosing two plaintext filesystem states that map to the same number of blocks and the same block IDs; see Definition 7.6. That fulfills the requirements in Definition 7.6 for an attacker against a f-multi-block encryption scheme. Since C^* is IND-atk secure, Lemma 7.11 implies the IND-atk security of $\mathsf{CryFS}^\mathsf{C}$.

With Lemma 7.10, the same result is obtained for IND-CCA2 security under Assumption 2.■

7.4 Integrity

In this section, we show that CryFS fulfills the integrity goals. Similar to the last section, we show in Theorem 2 that if the blocks are encrypted with an INT-PTXT secure scheme C and Assumption 1 holds, then $\mathsf{CryFS}^\mathsf{C}$ is INT-PTXT secure. The same holds for INT-CTXT security under the weaker Assumption 2. To prove this theorem, we build a chain of lemmata showing the following:

$$\mathsf{C}\ \text{INT-atk} \overset{7.12}{\Longrightarrow} \mathsf{C}'\ \text{INT-atk} \overset{7.13/7.15}{\Longrightarrow} \mathsf{C}^*\ \text{INT-atk} \overset{7.16}{\Longrightarrow} \mathsf{C}^\mathsf{f}\ \text{INT-atk} \overset{\text{Th.2}}{\Longrightarrow} \mathsf{CryFS}^\mathsf{C}\ \text{INT-atk}$$

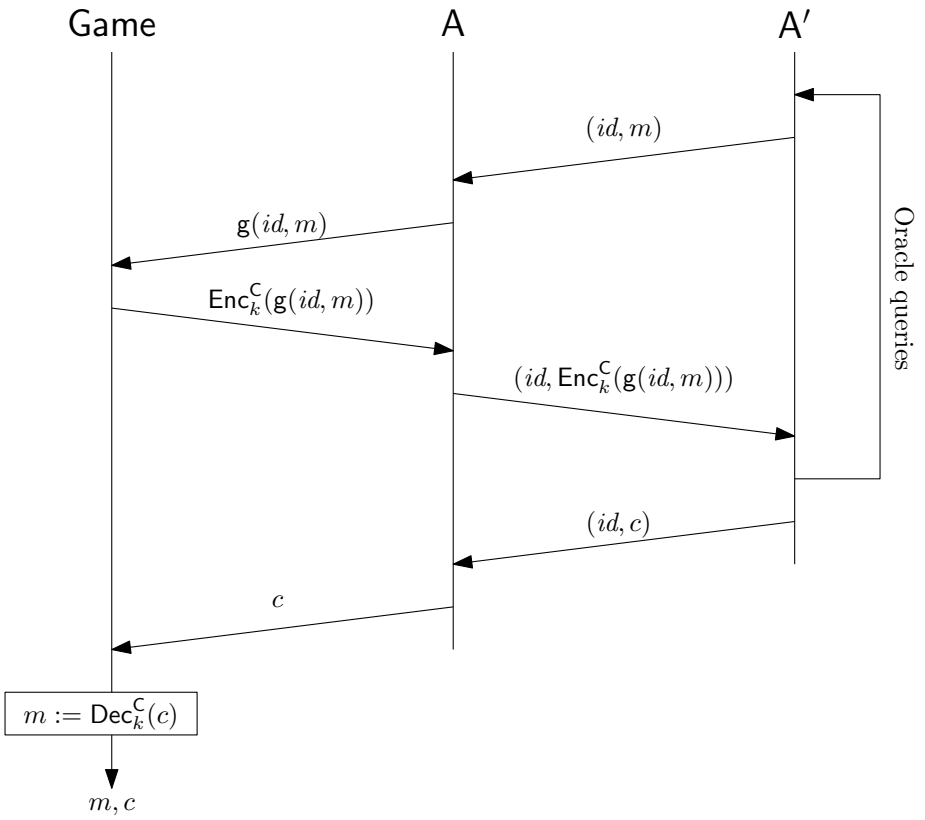


Figure 7.4 Proof for block integrity. An attacker A against C is built using an attacker A' against C'. As defined for block encryption, it encrypts a concatenation g of the block ID and data.

Throughout this section, it is $\mathsf{atk} \in \{\mathsf{PTXT}, \mathsf{CTXT}\}$.

Lemma 7.12 (C is INT-atk $\Rightarrow$ C' is INT-atk)
Let $\mathsf{C} = (\mathsf{Enc}_k^{\mathsf{C}}, \mathsf{Dec}_k^{\mathsf{C}})$ *be a symmetric encryption scheme that is* INT-atk *secure. Let* $\mathsf{C}' = (\mathsf{Enc}_k^{\mathsf{C}'}, \mathsf{Dec}_k^{\mathsf{C}'})$ *be the corresponding block encryption scheme. Then* C' *is* INT-atk *secure.*

Proof Let A' be an attacker who wins the INT-atk game against C'. Then the attacker A as shown in Figure 7.4 wins the INT-atk game against C.

A' generates a ciphertext (id, c) which, since A' is successful, decrypts successfully. Definition 7.2 specifies that (id, c) can only decrypt successfully if c decrypts successfully. That is, A sent a successfully decrypting ciphertext c to the game.

The oracles for A' are built from the oracles for A according to Definition 7.2.

In case of INT-PTXT security, we also have to show that $\mathsf{Dec}_k^{\mathsf{C}}(c)$ was never input to the oracle of A. Let $(id, p) := \mathsf{g}^{-1}(\mathsf{Dec}_k^{\mathsf{C}}(c))$. Because g is injective, we know that the only case in which $\mathsf{Dec}_k^{\mathsf{C}}(c) = \mathsf{g}((id, p))$ was input for the oracle of A is if (id, p) was input for the oracle of A'. In this case, $(id, p) = \mathsf{g}^{-1}(\mathsf{Dec}_k^{\mathsf{C}}(c)) = \mathsf{Dec}_k^{\mathsf{C}'}((id, c))$ was input for the oracle of A', which is a contradiction to A' being successful. Since A' is successful, the oracle for A never had $\mathsf{Dec}_k^{\mathsf{C}}(c)$ as oracle input and the INT-PTXT oracle condition for A is fulfilled.

In case of INT-CTXT security, we also have to show that c was never output of the oracle of A. We know that (id, c) was never oracle output for A', because otherwise A' would not have been successful. Assume there was an oracle output (id^*, c) with $id^* \neq id$. Then (id^*, c) has to decrypt successfully, which means that in Definition 7.2, we have $\mathsf{id}(\mathsf{g}^{-1}(\mathsf{Dec}_k^{\mathsf{C}}(c))) = id^* \neq id$. That is, (id, c) cannot have decrypted successfully which leads to a contradiction. So there never was any oracle output (id^*, c). That is, the oracle for A never had c as oracle output and the INT-CTXT oracle condition for A is fulfilled. ∎

The following Lemma 7.13 shows that with Assumption 2, INT-CTXT security for multi-block encryption follows from INT-CTXT security for block encryption.

Lemma 7.13 (Assumption 2 and C is INT-CTXT ⇒ C* is INT-CTXT)
Let $\mathsf{C} = (\mathsf{Enc}_k^{\mathsf{C}}, \mathsf{Dec}_k^{\mathsf{C}})$ *be a symmetric encryption scheme which is* INT-CTXT *secure. Let* $\mathsf{C}^* = (\mathsf{Enc}_k^{\mathsf{C}^*}, \mathsf{Dec}_k^{\mathsf{C}^*})$ *be the corresponding multi-block encryption scheme. Then, if Assumption 2 holds,* C^* *is* INT-CTXT *secure.*

Proof Since C is INT-CTXT secure, the block encryption scheme C' is INT-CTXT secure; see Lemma 7.12. So it is enough to reduce an attacker against C^* to an attacker against C'.

Let A^* be an attacker who wins the INT-CTXT game against C^*. Then the attacker A' as shown in Figure 7.5 loses the INT-CTXT game against C' only with negligible probability:

A^* sends a ciphertext $\widetilde{c} = \{c_1, \ldots, c_n\}$ to A'. Attacker A' selects any element c_1, and passes it to the game. Since A^* wins the game, we know $\mathsf{Dec}_k^{\mathsf{C}^*}(\{c_1, \ldots, c_n\}) \neq \perp$ which implies $\mathsf{Dec}_k^{\mathsf{C}}(c_1) \neq \perp$ by Definition 7.3.

The encryption oracle for A^* can be built using the encryption oracle for A'. We also have to show that c_1 was never output of the oracle of A'. We know that $\{c_1, \ldots, c_n\}$ was never output of the oracle of A^* and with Assumption 2, $\{c_1, c_2', \ldots, c_m'\}$ cannot have been output of the oracle of A^*. That is, there is no oracle query of A^* which caused the A' oracle to output c_1. ∎

As shown in the following Lemma 7.14, Assumption 2 is not enough to prove the same for INT-PTXT security. However, in Lemma 7.15, we show that Assumption 1 is enough to show that INT-PTXT security of multi-block encryption follows from INT-PTXT security of block encryption. Together with Lemma 7.7, this implies that Assumption 2 is strictly weaker than Assumption 1.

Lemma 7.14 (Assumption 2 and C is INT-PTXT ⇏ C* is INT-PTXT)
Let $\mathsf{C} = (\mathsf{Enc}_k^{\mathsf{C}}, \mathsf{Dec}_k^{\mathsf{C}})$ *be a symmetric encryption scheme which is* INT-PTXT*, but not* INT-CTXT *secure. Let* $\mathsf{C}^* = (\mathsf{Enc}_k^{\mathsf{C}^*}, \mathsf{Dec}_k^{\mathsf{C}^*})$ *be the corresponding multi-block encryption scheme. Even if Assumption 2 holds,* C^* *is not* INT-PTXT *secure.*

Proof We build a successful INT-PTXT attacker A^* against C^*.

Since C is not INT-CTXT secure, there exists an attacker A who successfully fakes a message c with $\mathsf{Dec}_k^{\mathsf{C}}(c) = m \neq \perp$. The encryption oracle for this attacker A can be built out of the oracle for A^* as follows: Given an oracle query q, we ask the encryption oracle of A^* for $\mathsf{Enc}_k^{\mathsf{C}^*}(\{q\}) = \{c\}$ and return c to A.

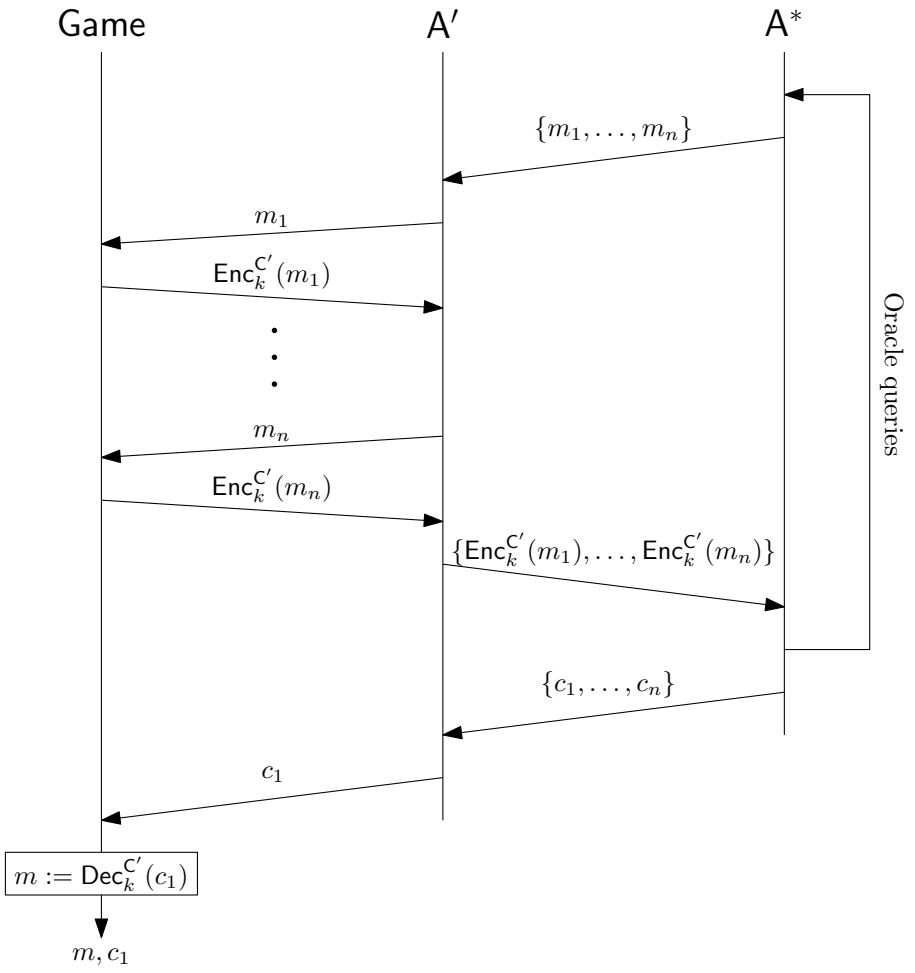


Figure 7.5 Proof for multi-block integrity. An attacker A' against C' is built using an attacker A^* against C^*. Oracle queries are answered by calling the oracle for each element in the set. In the end, one of the ciphertexts from the faked set is forwarded to the game.

The attacker A^* calls A two times and gets the faked messages $c_1 = \mathsf{Enc}_k^{\mathsf{C}}(m_1), c_2 = \mathsf{Enc}_k^{\mathsf{C}}(m_2)$ returned. A^* then returns $\{c_1, c_2\}$ to the game.

Let A be successful both times. That is, c_1 and c_2 were never oracle output of the oracle of A and therefore $\{c_1\}$ and $\{c_2\}$ were never oracle output of the oracle of A^*. That is, Assumption 2 cannot be applied and the attacker is allowed to output $\{c_1, c_2\}$. Because $\{c_1, c_2\}$ was never oracle output, A^* is successful as well. ∎

Lemma 7.15 (Assumption 1 and C is INT-PTXT ⇒ C* is INT-PTXT)
Let $\mathsf{C} = (\mathsf{Enc}_k^{\mathsf{C}}, \mathsf{Dec}_k^{\mathsf{C}})$ *be a symmetric encryption scheme which is* INT-PTXT *secure. Let* $\mathsf{C}^* = (\mathsf{Enc}_k^{\mathsf{C}^*}, \mathsf{Dec}_k^{\mathsf{C}^*})$ *be the corresponding multi-block encryption scheme. Then, if Assumption 1 holds,* C^* *is* INT-PTXT *secure.*

Proof Since C is INT-PTXT secure, the block encryption scheme C' is INT-PTXT secure; see Lemma 7.12. So it is enough to reduce an attacker against C^* to an attacker against C'.

Let A^* be an attacker who wins the INT-PTXT game against C^*. Then the attacker A' as shown in Figure 7.5 loses the INT-PTXT game against C' only with negligible probability:

A^* sends a ciphertext $\widetilde{c} = \{c_1, \ldots, c_n\}$ to A'. Attacker A' selects any element c_1, and passes it to the game. Since A^* wins the game, we know $\mathsf{Dec}_k^{\mathsf{C}^*}(\{c_1, \ldots, c_n\}) \neq \perp$ which implies $\mathsf{Dec}_k^{\mathsf{C}}(c_1) \neq \perp$ by Definition 7.3.

The encryption oracle for A^* can be built using the encryption oracle for A'. We also have to show that p_1 was never input for the oracle of A'. We know that $\mathsf{Dec}_k^{\mathsf{C}^*}(\{c_1, \ldots, c_n\}) = \{p_1, \ldots, p_n\}$ was never input for the oracle of A^*. It can only happen that p_1 is input for the oracle of A' if there is a $\{p_1, p_2', \ldots, p_r'\}$ that was input for the oracle of A^*. Since A^* is successful, this cannot happen according to Assumption 1. ∎

Since CryFS maps a filesystem state to a set of blocks before encryption, we have to show that this does not hurt integrity. This is done by the following Lemma 7.16.

Lemma 7.16 (C^* is INT-atk $\Rightarrow$ C^{f} is INT-atk)
Let C^* be a multi-block encryption scheme which is INT-atk *secure. Let C^{f} be the corresponding f-multi-block encryption scheme. For simplicity of notation, let $\mathsf{f}^{-1}(\perp) := \perp$. Then, C^{f} is* INT-atk *secure.*

Proof Let A^{f} be an attacker who wins the INT-atk game against C^{f}. Then the attacker A^* as shown in Figure 7.6 wins the INT-atk game against C^*.

A^{f} sends a ciphertext $c = \mathsf{Enc}_k^{\mathsf{C}^{\mathsf{f}}}(p) = \mathsf{Enc}_k^{\mathsf{C}}(\mathsf{f}(p))$ to A, which A^* redirects to the game. Because A^{f} is successful, we know $\mathsf{Dec}_k^{\mathsf{C}^{\mathsf{f}}}(c) = p \neq \perp$, which implies $\mathsf{Dec}_k^{\mathsf{C}}(c) = \mathsf{f}(p) \neq \perp$. That is, the ciphertext sent by A^* to the game decrypts successfully.

The encryption oracle for A^{f} can be directly built from the encryption oracle of A^* by applying f before encrypting. In case of INT-PTXT security, we also have to show that $\mathsf{f}(p)$ was never input to the oracle of A^*. This could only happen if p was input to the oracle of A^{f}. Because of $\mathsf{Dec}_k^{\mathsf{C}^{\mathsf{f}}}(c) = p$, this is a contradiction to A^{f} being successful. The oracle condition for INT-PTXT security is fulfilled.

In case of INT-CTXT security, we also have to show that c was never output of the oracle of A^*. This could only happen if c was also output of the oracle of A^{f} which is a contradiction to A^{f} being successful. The oracle condition for INT-CTXT security is fulfilled. ∎

And finally, we can now show the integrity theorem for CryFS. $\mathsf{CryFS}^{\mathsf{C}}$ is INT-PTXT secure if used with an INT-PTXT secure symmetric encryption scheme C and Assumption 1 holds, and it is INT-CTXT secure for an INT-CTXT secure symmetric encryption scheme under the weaker Assumption 2.

Theorem 2 (CryFS Integrity)
If the blocks are encrypted using an INT-PTXT(INT-CTXT) *secure symmetric encryption scheme and Assumption 1 (Assumption 2) holds, then $\mathsf{CryFS}^{\mathsf{C}}$ is* INT-PTXT(INT-CTXT) *secure.*

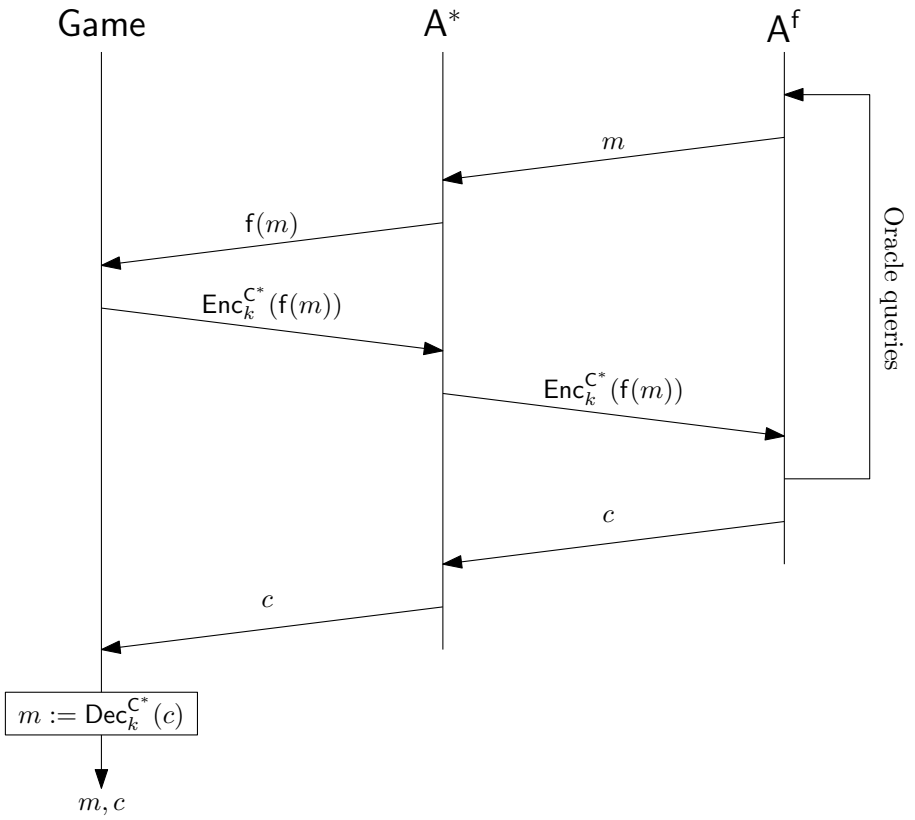


Figure 7.6 Proof for integrity under a function f. An attacker A^* against C^* is built using an attacker A^{f} against C^{f}. Oracle queries are answered by applying f to the plaintext before calling the oracle. The faked ciphertext is then forwarded to the game.

Proof Let $\mathsf{C} = (\mathsf{Enc}_k^{\mathsf{C}}, \mathsf{Dec}_k^{\mathsf{C}})$ be the INT-PTXT(INT-CTXT) secure encryption scheme encrypting the blocks of $\mathsf{CryFS}^{\mathsf{C}}$. $\mathsf{CryFS}^{\mathsf{C}}$ maps a filesystem state from $\mathbb{M}$ to a set of blocks from $\mathcal{P}(\mathbb{B}^*)$ and encrypts them using the f-multi-block encryption scheme C^{f}; see Definition 7.5. Since C is INT-PTXT(INT-CTXT) secure and Assumption 1 (Assumption 2) holds, C^* is INT-PTXT(INT-CTXT) secure as well; see Lemma 7.15 (Lemma 7.13). With Lemma 7.16, we get the INT-PTXT(INT-CTXT) security of $\mathsf{CryFS}^{\mathsf{C}}$. ∎

Note on INT-atk

In this section, the INT-atk notion as introduced in Section 3.1.3 has been used. In earlier works [BN08], they defined INT-atk relative to two constants q_d and q_e restricting the maximal number of queries to either oracle. In the notion used in this thesis, the attacker can use the encryption oracle without restriction but does not have access to a decryption oracle. We show in the following that this does not weaken the security notion. It is in fact equivalent to allowing the attacker an arbitrary but fixed number of decryption oracle queries.

Lemma 7.17
If a scheme is secure against an INT-atk *attacker who is not allowed to use a decryption oracle, then for any fixed q_d, it is also secure against an* INT-atk *attacker who is allowed to send at most q_d queries to a decryption oracle.*

PROOF Let A_{q_d} be an INT-atk attacker who can send at most q_d decryption oracle queries and has non-negligible success probability $\mathsf{Adv}_{\mathsf{INT\text{-}atk}}(\mathsf{A}_{q_d})$. We build an attacker A_0 who does not have a decryption oracle and still non-negligible success probability.

The encryption oracle for A_{q_d} can be built from the encryption oracle for A_0. When A_{q_d} queries the decryption oracle, A_0 decides randomly with probability $\frac{1}{2}$ whether it tells A_{q_d} that the query did not decrypt successfully or whether it sends this query as challenge to the game and finishes the game. If A_{q_d} sends a challenge, A_0 forwards it to the game.

If the i-th decryption oracle query would have decrypted successfully, then attacker A_0 has a chance of $(\frac{1}{2})^i$ of picking this query to forward it as a challenge to the game and win. Thus, if any of the decryption oracle queries would have decrypted successfully, A_0 has a success probability of at least $\mathsf{Adv}_{\mathsf{INT\text{-}atk}}(\mathsf{A}_0) \geq (\frac{1}{2})^{q_d}$, because $i \leq q_d$. This is non-negligible, because q_d is a constant value. If none of the decryption oracle queries would have decrypted successfully, then attacker A_0 has a chance of at least $(\frac{1}{2})^{q_d}$ of not picking any oracle query, but instead forwarding the challenge from A_{q_d} to the game. So A_0 has a success probability of $\mathsf{Adv}_{\mathsf{INT\text{-}atk}}(\mathsf{A}_0) = (\frac{1}{2})^{q_d} \cdot \mathsf{Adv}_{\mathsf{INT\text{-}atk}}(\mathsf{A}_{q_d})$. This is also non-negligible, because $\mathsf{Adv}_{\mathsf{INT\text{-}atk}}(\mathsf{A}_{q_d})$ is non-negligible. Thus, in any case, $\mathsf{Adv}_{\mathsf{INT\text{-}atk}}(\mathsf{A}_0)$ is non-negligible. ∎

That is, not restricting the access to the decryption oracle at all is a stronger security notion. However, restricting the allowed number of queries to an arbitrary but fixed number is equivalent to restricting the allowed number of queries to zero.

7.5 Adaptive Security

This section contains general ideas for adaptive security. A detailed analysis is out of scope of this thesis and remains future work. In the model of adaptive security as introduced by Dirk Achenbach et al. [Ach+15], there is an encrypted *database*, which in the case of CryFS is the encrypted filesystem, and there are *queries* on that database, i.e. filesystem operations. An attacker can see the access patterns of query processing, i.e. which blocks are read or written. There are two important security goals to consider, database privacy and query privacy. Both are formulated as indistinguishability games.

7.5.1 Goals

The first goal is *database privacy*, where the attacker is kept from gaining information about the database content, i.e. the data stored in the filesystem. The attacker can choose two databases m_1, m_2 of which the game chooses one m_b unknown to the attacker. However, the attacker gets $\mathsf{Enc}(m_b)$. Then, the attacker can send queries q_i to an oracle, which runs q_i on $\mathsf{Enc}(m_b)$ and returns the access pattern to the attacker. Afterwards, the attacker has to guess b.

The second goal is *query privacy*, where the attacker is kept from gaining information about the queries run, i.e. the filesystem operation executed. The attacker can choose two queries q_1, q_2 of which the game chooses one q_b unknown to the attacker. Then, the attacker can send databases m_i to an oracle, which runs q_b on $\mathsf{Enc}(m_i)$ and returns $\mathsf{Enc}(m_i)$ and the access pattern to the attacker. Afterwards, the attacker has to guess b.

These security goals are quite hard to achieve and can be weakened by restricting the oracle access, e.g. to a certain number of queries.

7.5.2 Attacker Types

The definitions for database privacy or query privacy state that an attacker gets the access patterns of queries. In the case of CryFS, this is the access pattern of filesystem operations. Two attacker types are distinguished. A *write observer* is an attacker who gets a sequence of the IDs of the blocks that are written by a filesystem operation. Each sequence item also contains a flag to tell the attacker which kind of operation it represents, i.e. whether the block was added, deleted or modified. A *general observer* is an attacker who gets a sequence that additionally contains the blocks that are only read, together with a flag telling the attacker that the block was only read. Obviously, a general observer is more powerful and harder to protect against. In the case of CryFS, a write observer is closer to practice though, because read accesses happen locally and the synchronization server is only able to see modifications.

7.5.3 Adapting CryFS

To be secure in an adaptive scenario, the CryFS implementation for resizing blobs has to be modified. This section explains the changes needed to make the access pattern of blob resize operations not depend on the current blob size. The goal is that the same blob operation, i.e. resizing a blob by the same number of bytes or modifying the same data region, has the same write access pattern, even in two filesystems where this blob has a different size. The modifications explained here are not implemented in the reference implementation.

Shrinking Blobs

When the size of a blob decreases, a subtree is deleted from the blob tree. The number of blocks deleted depends on the size of that subtree, which in turn depends on the current size of the whole blob. It is one, if the new blob size needs a leaf less but still the same number of inner nodes. It is greater than one, if the system deletes some of the inner nodes as well. An attacker could choose two filesystems which contain the same file but with a different size, then shrink the file size (or just delete the whole file) and distinguish the filesystems by the number of blocks that were deleted.

This problem can be solved by not deleting the blocks of the subtree directly. Instead, the filesystem keeps the blocks in unmodified form in a free-list and deletes them at a later timepoint in bulk. They could also be reused when new blocks are needed. With this, shrinking a blob causes exactly one block to be modified—the inner node where the subtree was rooted now has a child less. All other blocks are unmodified. If after this operation the root node only has one child left, CryFS decreases the tree depth by overwriting the data in the root block with the data from its child and deleting the child block. Deleting the child is not a problem because of the free-list and writing to the root block is not a problem, since this is exactly the same block that is modified anyhow in the first step because the subtree is rooted here.. The free-list does not have to be stored on the server, because all clients can locally build it by traversing all blocks and finding the unreferenced ones.

Growing Blobs

When the size of a blob increases, a subtree is added to the blob tree. The size of the subtree depends on the current size of the blob. To not spoil this size to an attacker, the

system takes a subtree of fitting size from the free-list described above. With that, growing a blob causes exactly one block to be modified—the inner node where the subtree is added as a child. All other blocks are unmodified.

If the tree is a max-data tree, CryFS increased the depth of the tree by copying the root block to a new block and overwriting the root block with a new root block that is one level above and has the copied block as first child. This causes two blocks to be written and is distinguishable. To prevent that, either the blob ID could be allowed to change, which would allow to just add a new root block above without copying and overwriting the old one. Alternatively, a dummy write operation could be introduced by re-encrypting an arbitrary block with a new IV each time a growing operation would otherwise only write to one block.

Writing to a Blob

When writing to a blob and the write operation is inside the blob in both filesystems, then it also modifies the same number of blocks and is indistinguishable. If in both blobs the write operation is causing the blob to grow, then it is indistinguishable as described in the previous paragraph. Growing a blob causes one block to be modified and writing data to a blob leaf causes one block to be modified. So if a write operation in one of the filesystems causes a blob to grow, then there are two modified blocks. To solve this, on each write operation that does not cause a blob to grow, the system re-encrypts a random block with a new IV. With this, each write operation modifies exactly two blocks.

7.5.4 Database Privacy

Database privacy is about keeping an attacker from gaining information about the database content. In the case of CryFS, this means keeping the attacker from gaining information about the data stored in the filesystem while the attacker can see the access patterns of filesystem operations.

7.5.4.1 Readonly Queries

For readonly queries, there are no write accesses happening, so the system is secure against a write observer by definition. So in this section, only general observers who have the ability to see read access patterns as well are considered.

Reading Content

When in CryFS a file is read, the system first looks up the file. This access pattern is determined by the file path only. Afterwards, the system descends to the leaves of the file blob. This access pattern is determined by the file size only. So the access pattern for a read query is determined by file path and file size. When a read query is issued, the file path is issued as a parameter to it, so the file path causes the same lookup pattern in both databases, given the file exists in both databases. So to get database privacy for read queries, an attacker has to be restricted to read from a file that exists and has the same size in both databases. Once this is fulfilled, the attacker does not have to be restricted further. It can run an unlimited number of oracle queries.

An exception to this is a filesystem where symlinks are allowed. If the filesystem supports symlinks, the access pattern is not only dependent on the path issued in the read query, but also on the path stored in the symlink, which makes database privacy much harder to achieve.

Search

When searching for file content or for the name of a file, the system has to traverse and read all blocks anyhow, because there is no index. The order of processing the blocks is not inherently important to a search algorithm, so it can process them in an arbitrary order which gives an attacker no information about the content. So database privacy for search queries is easily fulfilled. On modern systems however, the search functionality is not implemented in the filesystem, but in the operating system, which then issues read request to the filesystem. The generation of these read requests is not under the influence of the filesystem implementation and can make two filesystems distinguishable.

7.5.4.2 Write Queries

For these queries, there are read and write accesses happening. So it is important to look at both, write observers and general observers.

Adding Files/Directories

When in CryFS a file or directory is added, the system first looks up the directory which should contain the new entry, adds an entry to this directory blob, and creates a new blob for the file/directory itself. So there is a number of read operations for the lookup, which is dependent on the depth of the path. Then, one blob is modified, which is indistinguishable because of modifications described in Section 7.5.3. At last, there is one blob added which is independent from the filesystem state. So to be secure against a write observer, it only has to be ensured that before the operation, the entry does not exist in either filesystem; otherwise the attacker could see that in one of the filesystems there is just an error raised and nothing modified.

If security against a general observer is needed, the attacker additionally has to be restricted to add an entry to a directory that has the same depth in both filesystems. In a filesystem without symlinks, this is trivially fulfilled, because the path is issued by the attacker as a query parameter. Adding symlink functionality breaks security. Furthermore, because a general observer sees how many nodes the system reads when descending to the leaves of the directory blob, the directory has to have the same size in both filesystems. The number of oracle queries an attacker is allowed to do does not have to be restricted.

Deleting Files/Directories

When in CryFS a file or directory is deleted, the system first looks up the directory which contains this entry, deletes the entry from there, and deletes the blob containing the entry itself. Removing the entry from the directory blob is indistinguishable because of the modifications described in Section 7.5.3. The same is true for deleting the blob containing the entry itself. So to be secure against a write observer, the attacker only has to be restricted to deleting an entry that actually exists in both filesystems.

As before, a filesystem with symlinks is not be secure against a general observer. For a filesystem without symlinks to be secure against a general observer, the attacker has to be restricted to deleting entries from a directory that has the same size in both filesystems, because the attacker can see how many blocks the system reads when descending to the leaf of the directory blob. The number of oracle queries an attacker is allowed to do does not have to be restricted.

Writing Content

As before, the attacker has to be restricted to write to a file that either does or does not exist in both filesystems. Then, the system is secure against a write observer because writing the same data region in a blob causes the same number of blocks to be modified, independent of the concrete filesystem state. If the write operation grows the blob, the modifications described in Section 7.5.3 take care that the accesses are indistinguishable.

As before, a filesystem without symlinks is also secure against a general observer, because the lookup pattern is dependent on the path only. A filesystem with symlinks is not be secure against a general observer.

7.5.5 Query Privacy

Query privacy is about keeping an attacker from gaining information about the database queries. In the case of CryFS case, this means keeping the attacker from gaining information about the filesystem operations run while they can see the access patterns of these operations. This is a very interesting problem, as it is not easily achieved for a filesystem. We leave this for future work.

7.5.6 Conclusion

With some modifications to the filesystem implementation, Database Privacy can be achieved for CryFS. If the adversary is on server side, it is a write observer that can only see write operations, which makes read operations secure by definition. With the described modifications, CryFS is secure against such a write observer. Protecting against a general observer that can also see read operations is possible, but the described solution only works if the filesystem does not support symlinks. The number of oracle queries does not have to be restricted.

7.6 Summary

CryFS is IND-CPA secure if the underlying block cipher is IND-CPA secure. The same is true for IND-CCA1 security. For IND-CCA2 security, the same is proven under an additional integrity assumption.

It is also proven that INT-CTXT security of CryFS follows from INT-CTXT security of the underlying block cipher under the same assumption. If the underlying block cipher only fulfills INT-PTXT security, a slightly stronger assumption is needed to show INT-PTXT security of CryFS.

The assumptions made allow to keep a simpler stateless formalization, although CryFS itself implements integrity in a stateful way. Since the assumptions are close to the actual integrity implementation, they are justified and we believe them fulfilled.

8. Conclusion

In this thesis, we analyzed different existing solutions for secure cloud storage. To the best of our knowledge, none is secure for cloud storage and at the same time easy enough to be used in practice. We introduced CryFS, a cryptographic filesystem that is provably secure, easy to use, and works well together with cloud storage providers. While the primary target is to work together with third party synchronization clients as provided by providers like Dropbox, it can easily be modified to use other synchronization software like Rsync or Unison, or even to work without a local copy of ciphertexts and directly store the blocks remotely, for example on NFS or Amazon S3. To store its data, CryFS splits the filesystem data into fixed size blocks using balanced left-max-data trees. We introduced this kind of trees and showed that they are well suited for splitting resizeable blobs into fixed size blocks. They are proven to have very little space overhead, i.e. 0.05% in the reference implementation. We described algorithms for random access, querying the size of blobs and resizing them, proved the correctness of these algorithms and showed that these operations are fast.

We discussed different design options for how directory structure could be stored, and showed their advantages and disadvantages. All except for the real directory approach are based on blocks, which makes the integrity implementation described in this thesis work. It is interesting future work to implement integrity in the real directory approach. Analyzing the different options leads to the conclusion that a parent pointer approach in the variant with directory blobs and a cache is optimal, but a plain directory blob approach is easier to implement. We also provided some ideas on how partial shareability could be implemented in the different design options and refer to future work for looking into how attribute based encryption could make this simpler.

We implemented CryFS and will continue to maintain and improve this implementation. However, experiments have shown the performance of the current implementation to be already very good, allowing CryFS to be used in practice. The current implementation uses directory blobs, but we plan to offer an implementation with the parent pointer design in future. It also lacks some of the integrity features described in this thesis. It stores the block ID in the block header and uses GCM as an authenticated cipher, but it does not implement block version counters or an integrity-checked list of deleted blocks yet. We

are planning to add these in future. We also described the software architecture of the implementation and showed how we implemented it to be fast and secure. A reference explains the space layout and can be used to implement other software working with CryFS encrypted data.

CryFS as described in this thesis is secure and provably achieves confidentiality and integrity of file contents, file metadata and directory structure. It is transparent and easy to use. In their daily workflow, the user works directly with plaintext files while ignoring that it is backed by cryptography. The security of CryFS has been proven in a game-based approach. After introducing the basic security notions, we proved that file contents, file metadata and directory structure are IND-atk secure if the block cipher used to encrypt the blocks is IND-atk secure. For integrity, we showed the same for INT-atk security under an additional assumption. We introduced this assumption, because the actual integrity measures implemented are stateful, but the proofs are simpler when a stateless system is assumed. We reasonably believe the assumption to be true because it is closely connected to the integrity measures implemented, but this is not a formal proof. In future work, it could be possible to prove the same integrity notions with an even weaker assumption or maybe even to use a stateful model without such assumptions to prove integrity with. A brief look at adaptive security showed that database privacy is feasible, but needs future work. Query privacy is harder to achieve and also left to future work.

Overall, CryFS is a provably secure filesystem for cloud storage, that is fast, easy to use, and works well together with third party cloud storage providers.

Bibliography

[Ach+15] Dirk Achenbach, Matthias Huber, Jörn Müller-Quade, and Jochen Rill. *Closing the Gap: A Universal Privacy Framework for Outsourced Data*. Accepted for BalkanCryptSec 2015, https://conferences.matheo.si/conferenceDisplay.py?confId=16. 2015.

[Bel+97] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. "A concrete security treatment of symmetric encryption". In: *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*. Oct. 1997, pp. 394–403. DOI: 10.1109/SFCS.1997.646128.

[Bel+98] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. "Relations among notions of security for public-key encryption schemes". English. In: *Advances in Cryptology - CRYPTO '98*. Ed. by Hugo Krawczyk. Vol. 1462. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998, pp. 26–45. ISBN: 978-3-540-64892-5. DOI: 10.1007/BFb0055718.

[Bla00] John R Black Jr. "Message authentication codes". PhD thesis. UNIVERSITY OF CALIFORNIA DAVIS, 2000.

[Bla93] Matt Blaze. "A cryptographic file system for UNIX". In: *Proceedings of the 1st ACM conference on Computer and communications security*. ACM. 1993, pp. 9–16.

[BM95] Olaf Burkart and Bernhard Möller. *Mathematics of Program Construction: Third International Conference, MPC '95, Kloster Irsee, Germany, July 17 - 21, 1995. Proceedings*. 1995.

[BN08] Mihir Bellare and Chanathip Namprempre. "Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm". English. In: *Journal of Cryptology* 21.4 (2008), pp. 469–491. ISSN: 0933-2790. DOI: 10.1007/s00145-008-9026-x.

[DD05] Ivan Damgård and Kasper Dupont. *Universally Composable Disk Encryption Schemes*. Sept. 2005.

[DR11] Thai Duong and Juliano Rizzo. *Here Come The ⊕ Ninjas*. May 2011.

[Dwo07] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. 2007.

[Ewi] Larry Ewing. https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation. Created with The GIMP.

[Fou+08] Pierre-Alain Fouque, Gwenaëlle Martinet, Frédéric Valette, and Sébastien Zimmer. "On the Security of the CCM Encryption Mode and of a Slight Variant". In: *Applied Cryptography and Network Security*. Springer. 2008, pp. 411–428.

[Gjø05] Kristian Gjøsteen. *Security notions for disk encryption*. Apr. 2005.

[Goy+06] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. "Attribute-based encryption for fine-grained access control of encrypted data". In: *Proceedings of the 13th ACM conference on Computer and communications security*. Acm. 2006, pp. 89–98.

[GS14] Konstantinos Giannakouris and Maria Smihily. *Cloud computing - statistics on the use by enterprises*. http://ec.europa.eu/eurostat/statistics-explained/index.php/Cloud_computing_-_statistics_on_the_use_by_enterprises. 2014.

[Hal05] Michael Austin Halcrow. "eCryptfs: An Enterprise-class Encrypted Filesystem for Linux". In: *Proceedings of the Linux Symposium, Volume One*. 2005, pp. 201–218.

[Hor14] Taylor Hornby. *EncFS Security Audit*. https://defuse.ca/audits/encfs.htm. Feb. 2014.

[HW13] Susan Hohenberger and Brent Waters. "Attribute-based encryption with fast decryption". In: *Public-Key Cryptography–PKC 2013*. Springer, 2013, pp. 162–179.

[KL08] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC cryptography and network security. 2008, pp. xviii + 534. ISBN: 1-58488-551-3.

[LCH13] Cheng-Chi Lee, Pei-Shan Chung, and Min-Shiang Hwang. "A Survey on Attribute-based Encryption Schemes of Access Control in Cloud Environments." In: *IJ Network Security* 15.4 (2013), pp. 231–240.

[Mia10] Quan-xing Miao. "Research and analysis on encryption principle of truecrypt software system". In: *Information Science and Engineering (ICISE), 2010 2nd International Conference on*. IEEE. 2010, pp. 1409–1412.

[Moe04] B. Moeller. *Security of CBC Ciphersuites in SSL/TLS: Problems and Countermeasures*. May 2004.

[MV04] David A. McGrew and John Viega. *The Security and Performance of the Galois/Counter Mode (GCM) of Operation*. Proceedings of INDOCRYPT 2004. S. 343-355. 2004.

[SR14] Heidi Seybert and Petronela Reinecke. *Survey on ICT (information and communication technology) usage in households and by individuals*. Catalogue number: KS-SF-14-016-EN-N. Short version: http://ec.europa.eu/eurostat/statistics-explained/index.php/Internet_and_cloud_services_-_statistics_on_the_use_by_individuals. 2014.

[Tea11] Ubuntu Privacy Remix Team. *Security Analysis of TrueCrypt 7.0 a with an Attack on the Keyfile Algorithm*. Tech. rep. Technical report (Aug. 14, 2011). https://www.privacy-cd.org/downloads/truecrypt_7.0a-analysis-en.pdf, 2011.

[Toa] Ray Toal. *Ordered Trees*. http://cs.lmu.edu/~ray/notes/orderedtrees/.