ScramFS file system symmetric encryption layer: security analysis summary

Ron Steinfeld

Faculty of IT, Monash University

Document version 1.0

Abstract. This document contains a summary of the security analysis of the ScramFS encrypted file system. It summarizes the ScramFS security properties, the main cryptographic mechanisms used to achieve those properties and the underlying security assumptions, and provides quantitative security estimates for typical attack scenarios and ScramFS parameters.

Table of Contents

1 Introduction

ScramFS is a software encryption layer for file systems. It is designed to sit between a caller application (to be called 'App' in the following) and a storage file system (to be called StFS in the following). The standard transparent filesystem interface exposed by ScramFS to the App contains unencrypted file names and file contents and will be called 'AppFS'. The file name and contents written by ScramFS to the StFS are encrypted and authenticated versions of the file names and contents written by the App to the AppFS interface exposed by ScramFS.

The design specification of ScramFS is detailed in a separate document [12]. The purpose of this document is to summarize the security properties ScramFS was designed to achieve, the main cryptographic mechanisms used to achieve those properties and the underlying cryptographic assumptions, and summarize the analysis of how these mechanisms prevent attacks against those properties, with estimated attack success probabilities.

2 ScramFS Security Properties

ScramFS is designed to satisfy two types of security requirements: file system integrity and file system privacy. This section provides a summary of these security requirements, and also security limitations.

2.1 File System Integrity

The ScramFS file system integrity requirements are intended to protect the integrity of information stored in the file system from an attacker having full read/write access to the storage system (StFS) and its communications with the ScramFS client instance, but assuming the attacker has no access to ScramFS client instance master key.

Integrity protected attributes summary. ScramFS is designed to protect the integrity of the following App file system attributes:

- File contents: the data in the body of the App files.
- File/directory names
- File properties: content length, modification time (time of last modification), creation time.

ScramFS is **not** designed to protect against the following type of integrity attacks:

- File roll-back: a StFS file is replaced with a previous stored version of that file. (*remark:* This limitation is due to the stateless ScramFS client software).
- File deletion/undeletion: a previously created ScramFS file is deleted/restored from/to StFS. (*remark:* This limitation is due to the functionality requirement for fast file updates independent of the rest of the directory contents).

Attack model summary. The attacker is allowed to interact with a ScramFS client instance containing a random ScramFS master key K, calling chosen ScramFS functions on behalf of the App with the function arguments chosen by the attacker, except for the master key. The attacker can observe all communication sent from the ScramFS client instance to the StFS storage system, and sends attacker controlled data on behalf of the StFS back to the ScramFS client. The attacker's goal is to get the ScramFS client instance to accept without detecting an error a file or directory

with some 'new' value (i.e. a value not previously set via an App function call to ScramFS) for one of the integrity protected attributes listed above (see below for a summary of specific integrity attack goals).

It is assumed that the attacker **cannot** access the client instance ScramFS master key or any internal memory of the ScramFS client. The attacker also **cannot** modify the ScramFS client code that implements the ScramFS functions. The attacker is also assumed to have bounded computation time and bounded number of training interactions and amount of training data. We require for integrity security that the success probability of such an attack is negligible.

Capabilities of integrity attackers within this model. Security in the considered integrity attack model rules out the feasibility of powerful attacks such as 'chosen plaintext' attacks, in which the attacker chooses many file and directory names and calls the ScramFS client with CreateFile and Open calls to obtain the corresponding encrypted file/directory names, in order to learn some information on the system key or be able to forge 'new' file/dir names. Similarly, it rules out attacks in which the attacker is allowed to choose file content data to be written to created files using Write calls to the ScramFS client, in order to help forge other 'new' or modified file content. Even further, it rules out the strongest form of active attack, namely 'chosen ciphertext' attacks, in which the attacker modifies previously encrypted file/dir names and file headers or content and makes Read calls to the ScramFS client to learn some useful information on the client secret key or help in breaking the integrity of the system. ScramFS is designed that even under these conditions, it should be infeasible to forge authentic 'new' encrypted file/directory names and file content that passes the ScramFS client integrity checks. Note, however, there are a few minor limitations to this 'ideal' integrity attack model that are noted below.

Goals of integrity attacks within this model. The considered integrity attack model covers a range of integrity attack scenarios. We summarize the main attack goal scenarios that are covered within this model (see Sec. 4 regarding how such attacks are prevented).

- *Forge file/dir name*: The attacker creates an encrypted filename dir/file on StFS. When the attacker calls ListDir to the ScramFS client on directory dir, a new plaintext filename filename is returned, which has not been previously created by a ScramFS call, and no integrity error is returned. Alternatively, when the encrypted file is opened with an Open call, no integrity error is detected by the ScramFS client.
- *Modify/Create file data contents or headers*: The attacker modifies or creates forged data content or headers of an encrypted file encdir1/encfile1 on StFS. When the file is opened with an Open call, and a Read call is made, no integrity error is detected, but a new content is read from the file, that has never been written to that file by a ScramFS function call.
- *Replace content of one file with content of another*: The attacker replaces some content/headers in encdir1/file1 with content/headers from encdir2/file2. Or rearrange order of blocks in a file.
- *Replace headers of one file with headers of another*: The attacker replaces some content/headers in encdir1/file1 with content/headers from encdir2/file2.

Integrity security limitations: integrity attacks not covered in this model.

In addition to the unprotected integrity attacks above, we summarize several minor attack scenarios that ideally could be considered forgeries, but are *not* protected against in ScramFS.

- *No integrity guarantee for* IsDir *and* IsFile: It is possible for an attacker to cause IsDir or IsFile to output 'true' even though the tested file/directory was never created with a CreateFile or Open call. This was done for efficiency reasons; the problem can be avoided if the client issues a

ListDir call to check if a desired file or directory exists in the directory listing instead of calling IsDir or IsFile.

– *'new' file/dir names that are not considered* ListDir *forgeries*: Ideally, any file/dir name dir/name that is returned with no integrity error by calling ListDir on directory dir, would be considered 'new' (and hence a ListDir forgery) if dir/name has never been created via one of the following functions: CreateFile, Open in write mode, MakeDir, Rename and RenameDir with destination argument dir/name. However, in the ScramFS integrity model, dir/name is also not considered a 'new' file/dir name forgery if dir/name was the argument to one of the following functions: IsFile, IsDir, Remove, ResumeWriteLoc, Rename and RenameDir with source argument dir/name. This is because, in all those functions the corresponding ScramFS function reveals the encrypted version of dir/name to StFS even though this file may not have been created. For instance, calling IsFile on a non-existing dir/name would allow StFS to create such a valid encrypted file name. This 'non ideal' behaviour (adopted for efficiency reasons) is considered a minor issue because such forgeries for files would be detected by the ScramFS client as soon as the corresponding file is opened. Moreover, exploiting them is difficult as it requires the attacker to issue a query to one of the above functions on behalf of the client at the forgery argument. Finally, the problem can be avoided if the client issues a ListDir call to check if a desired file exists in the directory listing before calling one of the above functions.

2.2 File System Privacy

The ScramFS file system privacy requirements are intended to protect the confidentiality of information stored in the file system from an attacker having full read/write access to the storage system (StFS) and its communications with the ScramFS client instance, but assuming the attacker has no access to ScramFS client instance master key.

It is assumed that the attacker **cannot** access the client instance ScramFS master key or any internal memory of the ScramFS client. The attacker also **cannot** modify the ScramFS client code that implements the ScramFS functions. The attacker is also assumed to have bounded computational time and bounded number of training interactions and amount of training data.

Private attributes summary. ScramFS is designed to protect the privacy of the following file system attributes:

– File contents: the data in the body of the App files.
– Exact file content **length**: the exact byte length of the data in the body of the App files (however, the length rounded up to a multiple of a content block size parameter *Lblkov* is leaked (*remark:* ScramFS hides the length of file content data by encrypting file content in blocks of size *Lblkov*, and padding the file content with dummy bytes up to the nearest multiple of *Lblkov* before encryption of the last file content block).
– File/directory name **contents**: the characters in the names of the App files and directories.
– Exact file/directory name **length**: the exact character length of the names of the App files and directories (however, the length **up to a multiple of a name block length parameter** *nameLengthTrim* is leaked (*remark:* ScramFS hides the length of file/dir names by padding the names with dummy characters up to the nearest multiple of *nameLengthTrim* before encrypting the name).
– ScramFS formatting (with respect to 'ciphertext only' attacks): information in StFS names and file contents specifying the version and format, such as the length of a file header ciphertext. Individual file/directory names and file contents should be indistinguishable from random data.

Leaked attributes summary. ScramFS is **not** designed to protect the privacy of the following (leaked) file system attributes:

- Directory tree structure DT (including the number of files per directory).
- File/dir list/read/write/rename access patterns DTAP.
- Internal file read/write access patterns FAP (including updated file headers, file content blocks, and file footer).
- Rounded file content **length** $L.F_R$: the file content byte length rounded up to the smallest multiple of *Lblkov* greater than the file length (only individually leaked via internal file access patterns).
- Rounded file/directory name **length** $L.n_R$: the file/dir name character length rounded up to the smallest multiple of *L.nblk* greater than the file/dir name character length. (only individually leaked via file/dir access patterns).
- File header, file content block, and file footer lengths. (only individually leaked via internal file access patterns).

Remark: The individual rounded file/dir name, rounded file content length and header and footer lengths are only leaked via the file access patterns to an attacker that can observe the client to server access patterns. However, 'ciphertext only' attackers that have access to only only a single version of each individual full encrypted file can only learn the sum of the lengths of the encrypted file headers, rounded content length and the rounded file/dir name length, due to the 'ScramFS Formatting' protected attribute.

Attack model summary. The attacker is allowed a training phase to interact with a ScramFS client instance containing the ScramFS master key, calling chosen ScramFS functions on behalf of the App with the function arguments chosen by the attacker, except for the master key. The attacker can observe all communication sent from the ScramFS client instance to the StFS storage system, and sends attacker controlled data on behalf of the StFS back to the ScramFS client. At the end of the training phase, the attacker's goal is to distinguish whether it was interacting with the real ScramFS client, or with an 'ideal functionality' ScramFS client. Informally, an 'ideal functionality' ScramFS client simulates the responses of the real ScramFS client (i.e. the encrypted file/dir names, and encrypted file headers, footers, and content blocks) using random bit strings that are independent of the private attributes listed above. The only information used by the 'ideal functionality' ScramFS client is the unprotected file system attributes listed above (e.g. the file access patterns and the length of the header and content blocks). Moreover, any calls of the attacker that violate integrity of the file system (as in the integrity attack model above, e.g. replacing an encrypted content block from one file into another file when making an Open call to the second file) are responded to with an integrity error in the ideal functionality ScramFS client. This models the ability of the attacker to mount active attacks against the privacy of the protected attributes, such as forms of chosen-ciphertext attacks.

It is assumed that the attacker **cannot** access the client instance ScramFS master key or any internal memory of the ScramFS client. The attacker also **cannot** modify the ScramFS client code that implements the ScramFS functions. The attacker is also assumed to have bounded computation time and bounded number of training interactions and amount of training data. We require for privacy security that the distinguishing advantage of such an attack is negligible. This requirement implies that it is infeasible for a privacy attacker to learn any partial information on the private attributes listed above, other than the information revealed by the leaked attributes listed above.

Goals of privacy attacks within this model. The considered privacy attack model covers a range of privacy attack scenarios. Here, we list the main goals of such attack scenarios (see Sec. 4 regarding how such attacks are prevented).

- *Distinguishing encrypted file/dir names from random strings*: The attacker creates (on behalf of the client) a number of directories and files using CreateFile or Open (in write mode) ScramFS calls, and can distinguish between the corresponding encrypted file/dir names and character-encoded (via the underlying StFS.BinToStrEncode function) and padded random bit strings.
- *Distinguishing encrypted file headers/footers from random strings*: The attacker creates (on behalf of the client) a number of directories and files using CreateFile or Open (in write mode) ScramFS calls, and can distinguish between the corresponding encrypted file headers and random bit strings of the same length.
- *Distinguishing encrypted file content blocks from random strings*: The attacker creates (on behalf of the client) a number of directories and files using CreateFile or Open (in write mode) ScramFS calls, and can distinguish between the corresponding encrypted file content blocks and random bit strings of the same length.

Capabilities of privacy attacks within this model. Security in the privacy attack model rules out the feasibility of powerful active attacks such as 'chosen plaintext' attacks, where the attacker learns encrypted file and directory names and tries to use them to learn some information on the encryption secret key or the names of some other encrypted files or directories. Similarly, the attacker is allowed to choose file content data to be written to created files using Write calls to the ScramFS client, and learn the encrypted file content, in order to help in learning something about the content of other encrypted files. Even further, the attacker is allowed to mount 'chosen ciphertext' attacks, in which the attacker modifies previously encrypted file/dir names and file headers or content and makes Read calls to the ScramFS client to learn some useful information on the client secret key. ScramFS is designed that even under these conditions, it should be infeasible to learn any partial information on the protected attributes. Note, however, there are a few minor limitations to this 'ideal' model that are noted below.

3 ScramFS Security Overview: Cryptographic Mechanisms and how they are used to achieve ScramFS Security Properties

This Section gives an overview of the main cryptographic primitives and mechanisms used in ScramFS and how they are used to achieve the desired ScramFS security properties. We also summarize the security properties of the underlying cryptographic primitives. More details on protocol attacks and quantitative security estimates are summarized in the following section. More details on the design can be found in the design specification document [12]. For definitions of standard cryptographic terms see any standard text on modern cryptography (e.g. [6,1]).

3.1 Level 0 Security: Cryptographic Building Blocks Security

- Block cipher $\mathsf{CIPH} = (\mathsf{E}, \mathsf{D})$, specified in ScramFS to be AES-256[9], with block length $len.CIPH.b = 128$ bit, and key length $len.CIPH.k = 256$ bit. The security properties of ScramFS and all its other cryptographic primitives and mechanisms (apart from the externally specified pseudo-random bit generator used to generate randomness in the system) are ultimately based on the

security of this block cipher as a cryptographic pseudorandom Permutation (PRP) / Function (PRF), i.e. the infeasibility for an attacker to distinguish the outputs of E (when E is keyed with a random key unknown to the attacker) from the outputs of a random function. For quantitative ScramFS security estimates later on, we use the following standard security estimate on the distinguishing advantage $\mathsf{Adv}_{\mathsf{E}}^{PRF}(q,T)$ of the attacker against the PRF security of E, assuming an attack run-time T and q queries to the function:

$$\mathsf{Adv}_{\mathsf{E}}^{PRF}(q,T) \leq q^2/2^{len.CIPH.b+1} + \mathsf{Adv}_{\mathsf{E}}^{PRP}(q,T) = q^2/2^{129} + \mathsf{Adv}_{AES-256}^{PRP}(q,T), \quad (1)$$

for the specified 128-bit block length, where $\mathsf{Adv}_{AES-256}^{PRP}(q,T)$ is the distinguishing advantage of the attacker against the PRP security of AES-256. Currently, the best known attack against the PRP security of (full) AES-256 is a brute-force key search, so for our security estimates against classical computing attacks we assume:

$$\mathsf{Adv}_{AES-256}^{PRP}(q,T) \leq T/2^{256}, \quad (2)$$

the attack run-time T is measured in units equal to the encryption computation time of AES-256 on one block. For example, as mentioned above, aiming at '128-bit' security, we will evaluate attack success probabilities / distinguishing advantage against attacks with off-line run-time $T = 2^{128}$, for which key search advantage against AES-256 $\mathsf{Adv}_{AE-256}^{PRP}(q,T) \leq 1/2^{128}$ is negligible. In later security estimates, q corresponds to some 'on-line' ScramFS attack parameter, related to the number of files or number of content blocks the attacker queries to be encrypted by the ScramFS client key (see later for more details), but is independent of the off-line attack run-time.

Since ScramFS is also designed to be secure in a 'post-quantum' scenario, we will also provide security estimates against quantum computing attacks. For these attacks, we make the reasonable assumption that the attacked ScramFS client runs on a classical computer, whereas the attack algorithm runs on a quantum computer, but interfaces with the attacked client classically. Currently, the best known quantum attack algorithm against the PRP security of (full) AES-256 is the application of Grover's quantum algorithm [4] to AES-256 key search. The success probability of Grover's algorithm after T quantum queries to a quantum oracle implementation of AES-256 is about $T^2/2^{256}$, so for our security estimates against quantum computing attacks we assume:

$$\mathsf{Adv}_{AES-256}^{PRP}(q,T) \leq T^2/2^{256}, \quad (3)$$

where, similarly to above, the attack run-time T is measured in units equal to the encryption computation time of AES-256 on one block on a *quantum* machine (conservatively, we neglect the additional cost of Grover's 'diffusion' operator).

– Authenticated Encryption With Associated Data (AEAD) scheme $\mathsf{AE} = (\mathsf{Enc}, \mathsf{Dec})$, as defined in [13,8], with key length $len.AE.k = 256$ bit, nonce (IV) length $len.AE.IV = 128$ bit, authentication tag length $len.AE.tag = 128$ bit. In ScramFS, this scheme is specified to be the GCM mode [8] using the AES-256 CIPH block cipher. ScramFS uses GCM to authenticate and encrypt the file data content (split into blocks), authenticate and encrypt file headers and footers that encrypt file information and file content encryption keys, and authenticate and encrypt file integrity information. In addition, the GCM AE scheme is used in a pseudorandom function mode to implement the VILPRF used in file and directory name encryption key derivation and as a primitive in the SIV file/dir name encryption algorithm. In these applications, the security of ScramFS is based on the following two security properties of GCM:

1 Privacy security (priv): the security of Enc as a pseudorandom function (PRF), i.e. indistinguishability of GCM ciphertexts from random strings, under chosen plaintext and associated data and chosen nonce (IV) attacks with non-repeating nonces, for a random GCM key unknown to the attacker. It is shown in [5,10] that GCM achieves the priv PRF security if the block cipher CIPH is secure as a pseudorandom permutation (PRP).
2 Authenticity security (auth): The authentication security of GCM, i.e. the infeasibility of forging 'new' GCM ciphertexts (not previously output by Enc) that pass the GCM decryption check, under chosen plaintext, chosen associated data, and chosen nonce (IV) attacks with non-repeating nonces, for a random GCM key unknown to the attacker. It is shown in [5,10] that GCM achieves the auth security if the block cipher CIPH is secure as a pseudorandom permutation (PRP).

These two properties directly imply the indistinguishability of ciphertexts from random and authenticity in a chosen ciphertext attack, also known as Authenticated Encryption with Associated Data (AEAD) security [13,8]. From the bounds on auth and priv security of GCM in [10][Th.2 and 3] we get the following bound for attack distinguishing advantage against AEAD security of GCM with block cipher CIPH, against a distinguishing attacker that runs in time T, makes a total number q encrypt/decrypt queries and L total encrypted/decrypted data length (in number of $len.CIPH.b$-bit blocks) and uses a maximum length L_N (in number of $len.CIPH.b$-bit blocks) of IVs per query:

$$\mathsf{Adv}^{AEAD}_{\mathsf{GCM}}(q, L, L_N, T) \leq \frac{(L+q+1)^2 + 2^5 \cdot q \cdot (L+q+1) \cdot (L_N+1) + q \cdot (L+1)}{2^{len.CIPH.b}} + \mathsf{Adv}^{PRP}_{\mathsf{E}}(L+q+1, T). \quad (4)$$

– Fixed input-length Pseudorandom function PRF, with key length $len.PRF.k$ bit, input length $len.PRF.in$ bit, and output length $len.CIPH.out$ bit. In ScramFS version 1, we fix PRF to be the E algorithm of CIPH, i.e. AES-256, with $len.PRF.k = 256$, $len.PRF.in = len.PRF.out = 128$. ScramFS relies on the PRF security of this function, equivalent to PRF security of AES-256 above.
– Variable input-length Pseudorandom function VILPRF operating on strings of bit length at least 1 bit and at most $len.VILPRF.inmax$ bits and with output length $len.VILPRF.out$ bit. In ScramFS version 1, we fix VILPRF to be a function derived from the GCM mode using AES-256 with the input provided as the IV, with $len.VILPRF.k = 256$, $len.VILPRF.inmax = 2^{64} - 1$ bit, and $len.VILPRF.out$=128. The PRF security of VILPRF can be estimated from the AEAD security of GCM-AES-256, using bound (4) above with L_N the maximum VILPRF input length and $L = 0$.
– Deterministic authenticated encryption with associated data scheme $\mathsf{DE} = (\mathsf{encrypt_s iv}, \mathsf{decrypt_s iv})$, as defined in [11], with key length $len.DE.k = 512$ bit, nonce/auth. tag (IV) length $len.DE.IV = 128$ bit. In ScramFS, this scheme is used to encrypt file and directory names. The deterministic authenticated encryption security (DAE) is defined similarly to AEAD above (measuring the advantage of distinguishing encrypted ciphertexts from random strings under a chosen ciphertext attack), except that encryption is deterministic, so no random IV is needed as input (it is generated deterministically from the input and key) and repeating plaintexts are not allowed to be encrypted in the distinguishing attack. In ScramFS, this scheme is instantiated with block cipher CIPH in the SIV DE mode [11], using the CIPH-GCM based VILPRF construction above for the variable input length pseudorandom function of SIV, and using CIPH in counter (CTR)

mode [7] as the underlying randomized encryption scheme. Using known results on the security of SIV mode [11][Th. 2, Cor. 4] and the security of CTR mode [1][Th. 5.3], we can estimate its DAE distinguishing advantage security, against a distinguishing attacker that runs in time T, and makes q encrypt/decrypt queries, L total encrypted/decrypted data length (in number of $len.CIPH.b$-bit blocks) and L_{nmax} maximum plaintext plus additional authenticated data length (in number of $len.CIPH.b$-bit blocks):

$$\mathsf{Adv}^{DAE}_{\mathsf{SIV}}(q, L, L_{nmax}, T) \leq \mathsf{Adv}^{AEAD}_{\mathsf{GCM}}(2q, 0, L_{nmax}, T) + \frac{4q^2 + 2 \cdot q \cdot L}{2^{len.CIPH.b}} + \mathsf{Adv}^{PRP}_{\mathsf{E}}(L + q + 1, T). \tag{5}$$

– Cryptographic Pseudorandom bit generator $\mathsf{PRG} = (\mathsf{init}, \mathsf{out})$ with random seeding initialization: This algorithm is used in ScramFS to generate pseudorandom nonces (IV) and keys, but is not specified in the ScramFS design. It can also be replaced with a source of true random bits. The ScramFS design assumes that it provides output bits indistinguishable from random and independent of any previous values (even if the ScramFS instance is restarted). In this analysis we assume these bits are perfectly random.

3.2 Level 1 Security: Cryptographic Mechanisms

Directory Tree Key Derivation Pseudorandom Functions. To achieve a security separation between different file system directories, ScramFS encrypts and authenticates file and directory names, using a directory-unique pseudorandom key derived deterministically from the directory path and the ScramFS master key using a directory tree listing key derivation function DerKey.Dirl. The security properties of ScramFS are based on the security of this key derivation function DerKey.Dirl as a pseudorandom function (PRF).

We summarize the security argument for this function. The function DerKey.Dirl is a built from the variable-input length pseudorandom function primitive VILPRF using a tree-based recursive structure, which is a variant of the well-known GGM construction of pseudorandom functions [3]. To each directory path node dir1/dir2 in the directory tree, the function DerKey.Dirl assigns a key $K_{dir.l}(\mathsf{dir1/dir2}) = \mathsf{VILPRF}(K_{dir.l}(\mathsf{dir1}), \mathsf{Con.Kdirluse.dirl} \| \mathsf{dir2})$, which is the evaluation of VILPRF on the 'child edge' name dir2 (with a prepended constant Con.Kdirluse.dirl) using the key $K_{dir.l}(\mathsf{dir1})$ assigned to the parent directory node. In addition to the children nodes corresponding to subdirectories of a given path, each path node dir1 in the tree has one more 'self intermediate listing key' child node associated with the intermediate listing key $K_{dir.l.Int}(\mathsf{dir1}) = \mathsf{VILPRF}(K_{dir.l}(\mathsf{dir1}), \mathsf{Con.Kdirluse.Int})$, which is the evaluation of VILPRF, with a different constant Con.Kdirluse.Int than for subdirectory children nodes, to ensure that this key is derived by querying VILPRF at a new point, so the resulting output is independent of the subdirectory children node keys. The intermediate listing key $K_{dir.l.Int}(\mathsf{dir1})$ is the key actually used to encrypt the names of files and directories in the parent directory dir1. Using a 'hybrid argument' similar to that used in the GGM security reduction [3], over all the nodes in each depth level of the tree, it can be shown that the pseudorandomness (PRF) security of VILPRF implies the pseudorandomness (PRF) security of DerKey.Dirl, for the key values on leaf nodes of the tree, i.e. the $K_{dir.l.Int}$ intermediate listing key nodes. Based on this, we give quantitative estimates for the PRF security of the $K_{dir.l.Int}$ leaf key outputs, against a distinguishing attacker that runs in time T, for a directory tree of total size N_{dir} nodes (directories), containing a total of N_f files over all directories, each with a maximum

length of L_{nmax} 128-bit blocks:

$$\mathsf{Adv}^{PRF}_{\mathsf{DerKey.dirl}}(N_{dir}, L_{nmax}, T) \leq \max_{\{N_d(i)\}_i} \sum_{i=1}^{N_{dir}} (\mathsf{Adv}^{PRF}_{\mathsf{VILPRF}}(N_d(i)+2, L_{nmax}, T) + \mathsf{Adv}^{PRF}_{\mathsf{E}}(4, T)), \quad (6)$$

where $N_d(i)$ denotes the number of files in the ith directory for $i = 1, \ldots, N_{dir}$ and the maximum is over all possible choices for $N_d(i)$'s subject to $\sum_i N_d(i) = N_{dir}$. We remark that since our upper bound estimate for $\mathsf{Adv}^{PRF}_{\mathsf{VILPRF}}(N_d(i)+2, L_{nmax}, T)$ is a polynomial $g(N_d(i))$ in $N_d(i)$ with non-negative coefficients and therefore satisfying $\sum_i g(N_d(i)) \leq g(\sum_i N_d(i)) + N_{dir} \cdot g(0)$, the maximum above can be upper bounded by $g(N_{dir}) + N_{dir} \cdot g(0)$ (which is how we evaluate this estimate in the following section).

In the planned extended future ScramFS version supporting a 'non-recursive key sharing' mode, to share listing access to a directory dir1 with some user, the intermediate listing key $K_{dir.l.Int}(\mathsf{dir1})$ for dir1 would be shared with that user. The above security argument on the pseudorandomness of the intermediate listing keys implies that even given a number of leaf node keys $K_{dir.l.Int}(\mathsf{dir1})$, it is infeasible for an attacking user to distinguish the remaining intermediate listing leaf node keys of the directory tree, corresponding to directories that were not shared with the user, including the intermediate listing keys of subdirectories of dir1. Alternatively, in a 'recursive key sharing' mode, to share listing access to a directory dir1 and all its subdirectories, the key $K_{dir.l}(\mathsf{dir1})$ for the internal tree node corresponding to dir1 would be shared with the user. In this case, a slight modification of the above security argument implies the pseudorandomness of the 'not shared' leaf node keys against an attacking user. Indeed, we may consider the pruned key derivation tree which prunes off the 'shared' subtree consisting of all children nodes of dir1 in the subtree rooted at dir1. The pruned tree has the key $K_{dir.l}(\mathsf{dir1})$ assigned to a leaf node, and previous argument implies the remaining leaf nodes are indistinguishable from random even given a number of such $K_{dir.l}(\mathsf{dir1})$.

ScramFS also uses another content key derivation function DerKey.Dirc to derive, for each directory dir1 in the directory tree, a key $K_{dir.c}(\mathsf{dir1})$ for encrypting the ACC header of files (containing an encrypted file content header encryption key K_{FCH} that in turn can be used to decrypt the FCH header and obtain the content encryption keys). The security of ScramFS relies on the pseudorandomness of this function. The function DerKey.Dirc is used to enforce access control to content of the directory dir1 separately from listing access. The function DerKey.Dirc is constructed in an exactly analogous way to DerKey.Dirl discussed above, except that, to ensure separation of listing and content access, its root node key $K_{dir.c}(/)$ is indistinguishable from a random key independent of the root listing key $K_{dir.l}(/)$, as those two keys are derived using the pseudorandom function VILPRF queried at different points, keyed by a master directory tree key K_{DT} that is pseudorandomly derived from the master file system key K. The same arguments (and estimates) for DerKey.Dirl above apply also to give the pseudorandomness security of DerKey.Dirc.

File/Dir Name Encryption. The file/dir name encryption algorithm encName encrypts and authenticates the name of a file or directory *name* within a directory *dir* under a SIV encryption key $K_{SIV}(dir) = (K_{dir.l.E}, K_{dir.l.A})$ derived from the intermediate listing key $K_{dir.l.Int}(dir)$. The latter listing key is in turn derived pseudorandomly from the listing root key $K_{DT.l}$ and the directory name *dir* using the directory tree key derivation pseudorandom function DerKey.Dirl. To hide the exact character length of *name*, encName pads the binary encoded *name* to a multiple of the name length block size parameter. The function also truncates long name ciphertexts to fit within the maximum character length of the storage system, storing the remainder name ciphertext in the TRN header of the file (or directory meta file). After applying SIV with the name header

fnhdr as additional authenticated data, an additional file name header encryption algorithm is used to encrypt the header (containing ScramFS formatting information) *fnhdr* by XORing it with a pseudorandom one-time pad $prn = \mathsf{VILPRF}(K_{dir.Int.l.format}(dir), IV.path)$ evaluated on the SIV IV/auth. tag $IV.Path$. The key $K_{dir.Int.l.format}(dir)$ is derived pseudorandomly similarly to the key derivation functions above.

From a security analysis point of view it is convenient to combine the pseudorandom key derivation function with $\mathsf{encName}$ and think of the combined name encryption algorithm, to be called $\mathsf{encName}'$ here, such that given the listing root key $K_{DT.l}$, directory name *dir* and file/dir name *name*, $\mathsf{encName}'$ internally derives $K_{SIV}(dir) = (K_{dir.l.E}, K_{dir.l.A})$ for *dir* and encrypts *name* using $\mathsf{encName}$ under $K_{SIV}(dir)$ to give the ciphertext C, and $\mathsf{decName}'$ works analogously given dir and C. The security properties of ScramFS are based on the deterministic authenticated encryption (DAE) security of this combined scheme $(\mathsf{encName}', \mathsf{decName}')$, defined as above. The DE security of the (combined) name encryption $\mathsf{encName}'$ can be shown using the pseudorandomness of the key derivation function $\mathsf{DerKey.DirI}$ above, combined with a hybrid argument over the directory keys corresponding to all queried directories *dir* to relate it to the DE security of the underlying $\mathsf{encName}$ and hence SIV algorithm. Also, the *fnhdr* decryption XOR operation mapping the encrypted *fnhdr* to the decrypted one preserves the integrity security, as it is one-to-one for a fixed $IV.Path$, and *fnhdr* and $IV.path$ are part of the authenticated data input.

Quantitatively, it gives the following distinguishing advantage security estimate for the file name encryption algorithm combined with key derivation ($\mathsf{encName}'$) against a distinguishing attacker that runs in time T, for a directory tree of total size N_{dir} nodes (directories), with a total of N_f files over all directories, each with a maximum bit-encoded name length of L_{nmax} 128-bit blocks, in terms of the security of SIV above:

$$\mathsf{Adv}^{DAE}_{\mathsf{encName}'}(N_{dir}, N_f, L_{nmax}, T) \leq \max_{\{N_f(i)\}_i} \sum_{i=1}^{N_{dir}} \mathsf{Adv}^{DAE}_{\mathsf{SIV}}(N_f(i), N_f(i) \cdot L_{nmax}, L_{nmax}, T) + \mathsf{Adv}^{PRF}_{\mathsf{DerKey.dirI}}(N_{dir}, L_{nmax}, T), \quad (7)$$

where $N_f(i)$ denotes the number of files in the ith directory for $i = 1, \ldots, N_{dir}$ and the maximum is over all possible choices for $N_f(i)$'s subject to $\sum_i N_f(i) = N_f$. We remark that since our upper bound estimate for $\mathsf{Adv}^{DAE}_{\mathsf{SIV}}(N_f(i), N_f(i) \cdot L_{nmax}, T)$ is a polynomial $f(N_f(i))$ with non-negative coefficients and therefore satisfying $\sum_i f(N_f(i)) \leq f(\sum_i N_f(i)) + N_{dir} \cdot f(0)$, the maximum above can be upper bounded by $f(N_f) + N_{dir} \cdot f(0)$ (which is how we evaluate this estimate in the following section).

We summarize the main security points on file dir/name encryption with respect to ScramFS security:

- *Integrity Security:* Integrity of encrypted file/dir names, thanks to the integrity security (implied by DAE security summarized above) of the underlying SIV deterministic encryption scheme, along with the pseudorandomness of the directory tree listing key derivation function (discussed above).
- *Privacy Security:* Encrypted file and dir names are indistinguishable from random strings of length equal to the rounded length of the file/dir name, thanks to the privacy security (implied by DAE security summarized above) of the underlying SIV deterministic encryption scheme, along with the pseudorandomness of the directory tree listing key derivation function (discussed

above), and the padding of file/dir names prior to encryption to hide exact name length. Security against active chosen ciphertext attacks is implied by the integrity security above.

– *Individual File Ciphertext-only Privacy Security for file name header:* Formatting headers of file and dir names are indistinguishable from random strings, thanks to the pseudorandom pad used to hide them in the name encryption algorithm. However, no privacy protection for this formatting information is claimed against interactive and active attacks that interact with the ScramFS client and observe file access patterns. For example, the truncation flag in the file name header, that indicates whether the name was truncated or not, is revealed by whether the ScramFS client opens the file to retrieve the file truncation (TRN) header or not, when file name decryption is performed (e.g. by a ListDir query at level 2).
– *Negligible Probability $IV.Path$ collisions within each directory.* The $IV.Path$ produced by name encryption for a given file is intended to serve as a unique 'file identifier' within each directory. It is used inside the file body to link all components of the file body encryption to the file name. This uniqueness of the file identifier is important to prevent various integrity attacks such as swapping a file body with a file name, as discussed later in this document. Moreover, the security of SIV is based on non-repeating IVs within each directory. Therefore, the security of SIV (and encName) implies that the probability of such $IV.Path$ collisions is negligible. The privacy (DAE) security of SIV ensures that its IV.Path auth. tags are indistinguishable from random bit strings of length len.IV bits. Thus, up to the negligible advantage of the attacker against the DAE security of encName′ as given in (7), the probability of $IV.Path$ collisions is at most the probability of such collisions with random IVs of length len.IV, namely

$$p_{collIVPath} \leq N_f^2/2^{\mathsf{len.IV}}, \tag{8}$$

over all N_f system files (over all directories). Since $IV.Path$ length len.IV = 128 bit is large, this collision probability is negligible (see next section for example cases). We note that such collisions can also cause a correctness issue (where a truncated long file name collides with another one in the same directory) as noted in the design specification document, but they occur only with negligible probability.

File Body Encryption. We summarize security of each component of file body encryption.

ACC header encryption. The file ACC header encryption algorithm CreateHdr.ACC (the corresponding decryption algorithm is verify.ACC) encrypts the file's content access key K_{FCH} used to decrypt the FCH file content header using the GCM AE scheme. The ACC header is encrypted using the intermediate directory content key $K_{dir.c.Int}(\mathbf{dir})$ for the file's parent directory **dir**. This key is derived from the directory content key $K_{dir.c}(\mathbf{dir})$ using the pseudorandom content directory key derivation function DerKey.Dirc. Similarly to the file name encryption algorithm, the AEAD security of the ACC header encryption algorithm encACC′ (and corresponding decryption algorithm) obtained by combining CreateHdr.ACC with the key derivation function DerKey.Dirc, can be shown based on the pseudorandomness of DerKey.Dirc and AEAD security of GCM.

Quantitatively, it gives the following distinguishing advantage security estimate for encACC′ against a distinguishing attacker that runs in time T, for a directory tree of total size N_{dir} nodes (paths), with a maximum of q_f files/subdirs. per directory each with a maximum bit-encoded name length of L_{nmax} 128-bit block and a maximum bit-encoded ACC plaintext/associated data length

of L_{ACCmax} 128-bit blocks, in terms of the security of GCM above:

$$\mathsf{Adv}^{AEAD}_{\mathsf{encACC}'}(N_{dir}, N_f, L_{ACCmax}, T) \leq \max_{\{N_f(i)\}_i} \sum_{i=1}^{N_{dir}} \mathsf{Adv}^{AEAD}_{\mathsf{GCM}}(N_f(i), N_f(i) \cdot L_{ACCmax}, 1, T) \quad (9)$$
$$+\mathsf{Adv}^{PRF}_{\mathsf{DerKey.Dirc}}(N_{dir}, L_{nmax}, T),$$

where $N_f(i)$ denotes the number of files in the ith directory for $i = 1, \ldots, N_{dir}$ and the maximum is over all possible choices for $N_f(i)$'s subject to $\sum_i N_f(i) = N_f$. Note that $L_{ACCmax} \leq 5$ blocks.

We summarize the main security points for ACC header encryption with respect to ScramFS security:

- *Integrity Security:* Integrity of encrypted file's content access key K_{FCH} is achieved, thanks to the integrity security (auth security, implied by AEAD security summarized above) of the underlying GCM authenticated encryption scheme used for ACC header encryption, along with the pseudorandomness of the directory tree content key derivation function (discussed above). Active attacks that replace ACC header of one file into the body of another file are prevented due to the inclusion of the unique (except for negligible probability, see file name encryption above) file identifier $IV.path$ in the authenticated 'associated data' input of the GCM scheme in the ACC encryption algorithm.
- *Privacy Security:* The privacy of the encrypted file content access key K_{FCH} encrypted in the ACC header is achieved, thanks to the privacy security (implied by AEAD security summarized above) of the underlying GCM authenticated encryption scheme used for ACC header encryption, along with the pseudorandomness of the directory tree content key derivation function (discussed above).
- *Fresh GCM nonces: Negligible Probability $IV.ACC$ collisions.* The security of the underlying GCM authenticated encryption scheme used for ACC header encryption also relies on having non-repeating nonces $IV.ACC$ used for each ACC header encryption. The IV.ACC nonces are chosen as fresh random strings of length $len.AE.IV = 128$ bit for each ACC header encryption. Therefore, the probability of $IV.ACC$ collisions is at most

$$p_{collIVACC} \leq N_f^2 / 2^{\mathsf{len.AE.IV}}, \quad (10)$$

  over all N_f system files. Since $IV.ACC$ length $\mathsf{len.AE.IV} = 128$ bit is large, this collision probability is negligible (see next section for example cases).
- *Negligible Probability ($IV.Path$, $T.ACC$) auth. tag collisions over distinct files in system.* The security of the FCH header encryption algorithm against active attacks on the file system (see FCH header encryption below) requires that the pair ($IV.Path$, $T.ACC$) is unique over all files in the system (not just within each directory), so only different versions of the same file will share the same ($IV.Path$, $T.ACC$) pair. The security of the name encryption (as discussed above) together with the security of the ACC header encryption, the pseudorandomness of the directory tree key derivation functions, and the independence of the listing and content root keys, implies that the pair ($IV.Path$, $T.ACC$) is indistinguishable from a pair of independent random bit strings for each file over the whole file system. Thus, up to the negligible advantage of the attacker against the DAE security of SIV and GCM and the underlying key derivation functions, the probability of ($IV.ACC$, $T.ACC$) pair collisions over all files in the system is at most

$$p_{collIVPathTACC} \leq N_f^2 / 2^{\mathsf{len.IV}+\mathsf{len.IV.AE}}, \quad (11)$$

over all N_f system files. Since $\mathsf{len.IV} + \mathsf{len.IV.AE} = 256$ bit is large, this collision probability is negligible (see next section for example cases).

FCH header encryption. The file FCH header encryption algorithm $\mathsf{CreateHdr.FCH}$ (the corresponding decryption algorithm is $\mathsf{decrypt.FCH}$) encrypts the file's content key K_{FC}, integrity key K_{FI}, and file parameters including file length $L.F$ and file creation/modification times. It uses the GCM AE scheme with encryption key K_{FCH} from the ACC header. Given the privacy of K_{FCH} provided by the ACC header encryption, the AEAD security of FCH header encryption is obtained from the AEAD security of GCM and a hybrid argument over all independent K_{FCH} file keys in the file system. Quantitatively, it gives the following distinguishing advantage security estimate for FCH header encryption against a distinguishing attacker that runs in time T, over a file system containing in total N_f files, with each file being updated into a maximum of q_{fv} file 'versions' (including all modifications to a file and all renamed versions of a file, all of which share the same K_{FCH}), with a maximum bit-encoded FCH header plaintext/associated data length of L_{FCHmax} 128-bit blocks, in terms of the security of GCM above:

$$\mathsf{Adv}^{AEAD}_{\mathsf{CreateHdr.FCH}}(N_f, q_{fv}, L_{FCHmax}, T) \leq N_f \cdot \mathsf{Adv}^{AEAD}_{\mathsf{GCM}}(q_{fv}, q_{fv} \cdot L_{FCHmax}, 1, T). \tag{12}$$

Note that $L_{FCHmax} \leq 10$ blocks.

We summarize the main security points on file FCH header encryption with respect to ScramFS security:

- *Integrity Security:* Similarly to ACC header encryption, integrity of FCH header encrypted information is achieved, thanks to the integrity security (auth security, implied by AEAD security summarized above) of the underlying GCM authenticated encryption scheme used for FCH header encryption. Active attacks that replace FCH header of one file into the body of another file (that is not a previous version of the other file; such 'file version rollback' attacks are not protected against in ScramFS) are prevented due to the inclusion of the unique (except for negligible probability, see file ACC header encryption above) file identifier $IV.path$ and the ACC header auth. tag $T.ACC$ pair in the authenticated 'associated data' input of the GCM scheme used in the FCH encryption algorithm.
- *Privacy Security:* The privacy of the encrypted file content access key K_{FCH} encrypted in the ACC header is achieved, thanks to the privacy security (implied by AEAD security summarized above) of the underlying GCM authenticated encryption scheme used for ACC header encryption.
- *Fresh GCM nonces: Negligible Probability $IV.FCH$ collisions.* Similarly to ACC header encryption, GCM security relies on non-repeating nonces $IV.FCH$ in each FCH header encryption with respect to the same K_{FCH} encryption key. The $IV.FCH$ nonces are chosen as fresh random strings of length $len.AE.IV = 128$ bit for each FCH header encryption, in every update of a file. Thus, the probability of $IV.FCH$ collisions is at most

$$p_{collIVFCH} \leq N_f \cdot q_{fv}^2 / 2^{\mathsf{len.AE.IV}}, \tag{13}$$

  over all N_f system files and at most q_{fv} created file versions (including renames and file updates) per file. Since $IV.FCH$ length $\mathsf{len.AE.IV} = 128$ bit is large, this collision probability is negligible (see next section for example cases).
- *Negligible Probability $T.FCH$ auth. tag collisions over file versions.* The FI segment integrity check security of ScramFS against replacement of an FCH block from one version of a file into

another (e.g. to allow the attacker to change the modification time) relies on FCH header auth. tags $T.FCH$ to be distinct for distinct versions of a file. The AEAD security of the FCH header encryption implies that the tags $T.ACC$ for distinct versions of a file are indistinguishable from independent random bit strings. Therefore, up to the negligible advantage of the attacker against the AEAD security of GCM, the probability of $T.ACC$ collisions over all files in the system is at most

$$p_{collTFCH} \leq N_f \cdot q_{fv}^2 / 2^{\mathsf{len.AE.tag}}. \tag{14}$$

Since $\mathsf{len.AE.tag} = 128$ bit is large, this collision probability is negligible (see next section for example cases).

FC content encryption. The file FC content encryption algorithm $\mathsf{encBlock}$ (the corresponding decryption algorithm is $\mathsf{decBlock}$) encrypts a file's content block of length $Lblk$ bytes. It uses the GCM AE scheme with encryption key $K_{FC}(ind)$ derived pseudorandomly with $\mathsf{DerKey.FCBlk}$ using the blockcipher E as a PRF keyed by K_{FC} on input the file block index ind. The AEAD security of FC encryption then is obtained by a hybrid argument over all file blocks in all files of the filesystem, from the pseudorandomness of E and the AEAD security of GCM.

Quantitatively, it gives the following distinguishing advantage security estimate for FC content encryption against a distinguishing attacker that runs in time T, over a file system containing in total N_{fc} content blocks read/written to all N_f files in the file system, with each file (and hence each file block) being updated into a maximum of q_{fv} file 'versions' (including all modifications to a file and all renamed versions of a file, all of which share the same K_{FC}), with a maximum FC block length L_{blkmax} 128-bit blocks and a maximum of N_{fcpf} FC blocks per file:

$$\begin{aligned}\mathsf{Adv}^{AEAD}_{\mathsf{encBlock}}(N_f, N_{fc}, N_{fcpf}, q_{fv}, L_{blkmax}, T) \leq N_{fc} \cdot \mathsf{Adv}^{AEAD}_{\mathsf{GCM}}(q_{fv}, q_{fv} \cdot L_{blkmax}, 1, T) \\ + \max_{\{N_{fc}(i)\}_i} \sum_{i=1}^{N_f} \mathsf{Adv}^{PRF}_{\mathsf{DerKey.FCBlk}}(N_{fc}(i), T),\end{aligned} \tag{15}$$

where $N_{fc}(i)$ denotes the number of FC blocks in the ith system file for $i = 1, \ldots, N_f$ and the maximum is over all possible choices for $N_f(i)$'s subject to $\sum_i N_{fc}(i) = N_{fc}$ and $N_f(i) \leq N_{fcpf}$ for all i.

We summarize the main security points on file FC content encryption with respect to ScramFS security:

- *Integrity Security:* Integrity of FC content blocks is achieved, thanks to the integrity security (auth security, implied by AEAD security) of GCM. Active attacks that replace FC blocks of one file into into the body of another file (that is not a previous version of the other file; such 'file version rollback' attacks are not protected against in ScramFS) are prevented due to per-file keys K_{FC} and the uniqueness of block auth. tags $T.FC.ind$. Reordering of blocks is prevented due to the (indistinguishable from random) independent keys used for distinct blocks.
- *Privacy Security:* The privacy of the encrypted file content blocks follows from the privacy security (implied by AEAD security summarized above) of the underlying GCM scheme.
- *Fresh GCM nonces: Negligible Probability $IV.FC.ind$ collisions.* Similarly to ACC header encryption, GCM security relies on non-repeating nonces $IV.FC.ind$ in each FC block over all versions of a given file, with respect to the same $K_{FC}(ind)$ key. The $IV.FC.ind$ nonces are chosen as fresh random strings of length $len.AE.IV = 128$ bit for each block encryption, in every update of a file. Thus, the probability of FC block IV collisions is at most

$$p_{collIVFC} \leq N_{fc} \cdot q_{fv}^2 / 2^{\mathsf{len.AE.IV}}, \tag{16}$$

over all N_{fc} file blocks and at most q_{fv} created file versions (including renames and file updates) per file. Since $IV.FCH$ length $\mathsf{len.AE.IV} = 128$ bit is large, this collision probability is negligible (see next section for example cases).

– *Negligible Probability $T.FC.ind$ auth. tag collisions over file versions.* The FI segment integrity check security of ScramFS against replacement of an FC content block from one version of a file into another, e.g. to allow the attacker to insert a block from an old file version into a new file version (which already has other new blocks), relies on FC block auth. tags $T.FC.ind$ to be distinct for distinct versions of a file. The AEAD security of the FC encryption implies that $T.FC.ind$ for distinct versions of a file are indistinguishable from independent random bit strings. Therefore, up to the negligible advantage of the attacker against the AEAD security of GCM, the probability of $T.FC.ind$ collisions over all N_{fc} file blocks and at most q_{fv} created file versions (including renames and file updates) per file in the system, is at most

$$p_{collFCT} \leq N_{fc} \cdot q_{fv}^2 / 2^{\mathsf{len.AE.tag}}, \tag{17}$$

Since $\mathsf{len.AE.tag} = 128$ bit is large, this collision probability is negligible (see next section for example cases).

FI footer encryption. The file FI footer integrity authenticated encryption algorithm $\mathsf{CreateFI}$ (the corresponding verification algorithm is $\mathsf{VerifyFI}$) verifies a GCM AE authentication tag on the list of file name, header and content auth. tags that serve as a unique (except with negligible probability) file 'version' identifier. It uses the GCM AE scheme with encryption key K_{FI} (from the FCH header). The auth integrity security of FI authentication is then obtained by a hybrid argument over all files of the filesystem, from the AEAD security of GCM.

Quantitatively, it gives the following distinguishing advantage security estimate for FI encryption against a distinguishing attacker that runs in time T, over a file system containing in total N_f files of total length N_{fc} (in number of FC blocks, each of length $Lblk$ bytes), with each file being updated into a maximum of q_{fv} file 'versions' (including all modifications to a file and all renamed versions of a file, all of which share the same K_{FI}), with a maximum of N_{fcpf} FC blocks per file:

$$\mathsf{Adv}^{AEAD}_{\mathsf{CreateFI}}(N_f, N_{fc}, N_{fcpf}, q_{fv}, T) \leq \max_{\{N_{fc}(i)\}_i} \sum_{i=1}^{N_f} \cdot \mathsf{Adv}^{AEAD}_{\mathsf{GCM}}(q_{fv}, q_{fv} \cdot (3 + N_{fc}(i)), 1, T), \tag{18}$$

where $N_{fc}(i)$ denotes the number of FC blocks in the ith system file for $i = 1, \ldots, N_f$ and the maximum is over all possible choices for $N_f(i)$'s subject to $\sum_i N_{fc}(i) = N_{fc}$ and $N_f(i) \leq Nfcpf$ for all i.

We summarize the main security points on file FI footer encryption with respect to ScramFS security:

– *Integrity Security:* Integrity of FI footer is achieved, thanks to the integrity security (auth security, implied by AEAD security) of GCM. Active attacks that replace FC blocks or FCH headers of one file into the body of another file (that is not a previous version of the other file; such 'file version rollback' attacks are not protected against in ScramFS) are prevented due to the uniqueness of the tag list that is authenticated by FI for each distinct file version (except with negligible probability collision events treated above).

– *Privacy Security:* The privacy of the encrypted FC block tag list (for single file contents indistinguishability from random) follows from the privacy security (implied by AEAD security summarized above) of the underlying GCM scheme.

– *Fresh GCM nonces: Negligible Probability IV.FC.ind collisions.* Similarly to ACC header encryption, GCM security relies on non-repeating nonces $IV.FI$ in each FC block over all versions of a given file, with respect to the same K_{FI} key. The $IV.FI$ nonces are chosen as fresh random strings of length $len.AE.IV = 128$ bit for each FI segment authentication, in every update of a file. Thus, the probability of FI IV collisions is at most

$$p_{collIVFI} \leq N_f \cdot q_{fv}^2 / 2^{\mathsf{len.AE.IV}}, \tag{19}$$

over all N_f files in the filesystem and at most q_{fv} created file versions (including renames and file updates) per file. Since $IV.FI$ length $\mathsf{len.AE.IV} = 128$ bit is large, this collision probability is negligible (see next section for example cases).

4 Level 2 Security: Attack Scenarios and why they are Prevented in ScramFS

4.1 Integrity Attacks

We summarize how different attack scenarios against the file system integrity are prevented in ScramFS based on the security of the underlying mechanisms, and estimate the success probabilities of those attack scenarios.

Attack scenario: Modification/Creation of a file/dir name. The attacker creates a new encrypted filename encdirname/encfilename on StFS. When the directory encdirname is listed with a ListDir call, a new plaintext filename filename is returned, which has not been previously created by a ScramFS call, and no integrity error is returned. Alternatively, when the file is opened with an Open call, no integrity error is detected.

How it is prevented in ScramFS: The deterministic authenticated encryption (DAE) security of the file/dir name encryption algorithm in ScramFS (see previous section) implies that the success probability of such file/dir name forgery attacks is negligible, assuming the pseudorandomness of the underlying AES-256 block cipher.

Quantitatively, suppose we modify the way we respond to the attacker's ScramFS calls by switching the file/dir name ciphertexts to random strings and rejecting 'new' encrypted file dir/name ciphertexts created by the attacker (as in the 'ideal' ScramFS client). Then the AEAD security of the combined key derivation and file/dir name encryption $\mathsf{encName}'$ (see previous section) implies that the attacker can distinguish between the new game and the original attack with at most negligible distinguishing advantage p_1, and hence (since in the new game, the attacker succeeds in getting the forged name accepted with zero probability) the file/dir name forging probability is at most p_1, where

$$p_1 \leq \mathsf{Adv}^{DAE}_{\mathsf{encName}'}(N_{dir}, q_f, L_{nmax}, T), \tag{20}$$

is negligible by (7). See scenario estimates below for examples.

Attack scenario: Forge header block or replace header of one file with header of another file: The attacker forges a header block or replaces headers in one file (say dir1/file1) with headers from another file (say dir2/file2).

How it is prevented in ScramFS: The authentication integrity security of the GCM-based file header encryption algorithm and the pseudorandomness of the directory tree key derivation implies that the success probability of such forgery attacks is negligible, assuming the pseudorandomness of the underlying AES-256 block cipher.

Quantitatively, we consider ACC and FCH headers in turn.

First, suppose we further modify the way we respond to the attacker's ScramFS calls from the scenario above by switching the ACC header ciphertexts to random strings and rejecting 'new' encrypted ACC header blocks for the file when opening the file (as in the 'ideal' ScramFS client). Then the AEAD security of the combined key derivation and ACC header encryption encACC', and the negligible probability of IV.Path collisions for files within the same directory (see previous section), implies that the attacker can distinguish between the new game and the previous one with at most negligible distinguishing advantage p_2, and hence (since in the new game, the attacker succeeds in getting the forged/replaced ACC header accepted with zero probability) this ACC header forging probability is at most $p_1 + p_2$, where

$$p_2 \leq \mathsf{Adv}^{DAE}_{\mathsf{encName}'}(N_{dir}, q_f, L_{nmax}, T) + p_{collIVACC} + p_{collIVPath}, \tag{21}$$

which is negligible by (7), (10) and (8). See scenario estimates below for examples. We note two subcases of this scenario:

1 A forged encrypted ACC header of file dir1/file1 is either completely new or has appeared before as a header of a file dir2/file2 in a different directory dir2 $\neq$ dir1 This subcase directly leads to a forgery against the integrity security of $\mathsf{encName}'$, or to a collision of ACC IVs, and is negligible as above by the AEAD security of $\mathsf{encName}'$ and negligible collision of ACC IVs.
2 The forged encrypted ACC header of file dir1/file1 was taken from another file file2 $\neq$ file1 in the same directory (dir2 = dir1). This also leads to a forgery against $\mathsf{encName}'$ but only if the $IV.Path$(dir1/file1) is different from $IV.Path$(dir1/file2) used in encrypting ACC header of file2. The probability that those auth. tags collide is at most p_{collIV}, so this subcase has probability at most $p_1 + p_2$ as above.

Next, we consider FCH header forgery scenarios. Suppose we further modify the way we respond to the attacker's ScramFS calls from the scenario above by switching the FCH header ciphertexts to random strings and rejecting 'new' encrypted FCH header blocks for the file when opening the file (as in the 'ideal' ScramFS client). Then the AEAD security of FCH header encryption encACC', and the negligible probability of $(IV.Path, T.ACC)$ pair collisions over all files in the system (see previous section), implies that the attacker can distinguish between the new game and the previous one with at most negligible distinguishing advantage p_3, and hence (since in the new game, the attacker succeeds in getting the forged/replaced FCH header accepted with zero probability) this FCH header forging probability is at most $p_1 + p_2 + p_3$, where

$$p_3 \leq \mathsf{Adv}^{AEAD}_{\mathsf{CreateHdr.FCH}}(N_f, q_{fv}, L_{FCHmax}, T) + p_{collIVFCH} + p_{collIVPathTACC}, \tag{22}$$

which is negligible by (12), (13) and (11). See scenario estimates below for examples. Due to the already considered name and ACC forgery scenarios above, we may assume that the forged FCH header for dir1/file1 was provided with encrypted names and ACC header from a previously created version of dir1/file1. We note three subcases of this scenario:

1 The forged FCH header of file dir1/file1 is either completely new or has appeared before as FCH header of a file dir2/file2 which is not a renamed or previous version of dir1/file1 (so it uses an independent FCH encryption key K_{FCH}). This subcase directly leads to a forgery against the integrity security of AEAD security of FCH header encryption, and is negligible by the AEAD security of CreateHdr.FCH.

2 The forged FCH header of file dir1/file1 was taken from another file file2 $\neq$ file1 in the same directory (dir2 = dir1) which is a renamed version of dir1/file1. In this case, the same K_{FCH} will be used to verify the forged FCH block in file1 as was used to create the forged FCH header in file2. However, this also leads to a forgery against AEAD security of CreateHdr.FCH, thanks to the inclusion of $IV.Path$(dir1/file1) in 'associated data' input of FCH header encryption, and the lack of collisions of $IV.Path$ within each directory in this game (see previous scenarios above).

3 The forged FCH header of file dir1/file1 was taken from another file file2 in a different directory (dir2 $\neq$ dir1), which is a renamed/moved version of dir1/file1. In this case, the same K_{FCH} will be used to verify the forged FCH block in file1 as was used to create the forged FCH header in file2. However, this also leads to a forgery against AEAD security of CreateHdr.FCH, except with negligible probability, thanks to the inclusion of the pair $(IV.Path, T.ACC)$ in 'associated data' input of FCH header encryption and the negligible probability $p_{collIVPathTACC}$ of collisions of $(IV.Path, T.ACC)$ pairs over all files in the system.

Attack scenario: Forge FI footer block or replace FI of one file with FI of another file: The attacker forges a FI footer block or replaces FI in one file (say dir1/file1) with FI from another file (say dir2/file2).

How it is prevented in ScramFS: Integrity of FI footer is achieved, thanks to the AEAD security of GCM. Active attacks that replace FC blocks or FCH headers of one file into the body of another file (that is not a previous version of the other file; such 'file version rollback' attacks are not protected against in ScramFS) are prevented due to the uniqueness of the tag list that is authenticated by FI for each distinct file version (except with negligible probability collision events treated above).

Quantitatively, suppose we further modify the way we respond to the attacker's ScramFS calls from the scenario above by switching the FI footer ciphertexts to random strings and rejecting 'new' encrypted FI footer blocks for the file when opening the file (as in the 'ideal' ScramFS client). Then the AEAD security of FI footer encryption CreateFI, and the negligible probability of ($IV.Path$, $T.ACC$, $T.FCH$) collisions over all distinct versions of files in the system (see previous section), implies that the attacker can distinguish between the new game and the previous one with at most negligible distinguishing advantage p_4, and hence (since in the new game, the attacker succeeds in getting the forged/replaced FI footer accepted with zero probability) this FI footer forging probability is at most $p_1 + p_2 + p_3 + p_4$, where

$$p_4 \leq \mathsf{Adv}^{AEAD}_{\mathsf{CreateFI}}(N_f, N_{fc}, N_{fcpf}, q_{fv}, T) + p_{collIVFI} + p_{collTFCH}, \tag{23}$$

which is negligible by (18), (19) and (14). See scenario estimates below for examples.

Due to the already considered name, ACC and FCH forgery scenarios above, we may assume that the forged FCH header for dir1/file1 was provided with encrypted names and ACC and FCH header from a previously created version of dir1/file1. We note two subcases of this scenario:

1 The forged FI footer of file dir1/file1 is either completely new or has appeared before as FI footer of a file dir2/file2 which is not a renamed or previous version of dir1/file1 (so it uses an independent FCH encryption key K_{FCH}). This subcase directly leads to a forgery against the integrity security of AEAD security of FI footer encryption thanks to the inclusion of the pair $(IV.Path, T.ACC)$ in 'associated data' input of FI footer encryption (in the case of taking from a different file, ($IV.Path$, $T.ACC$) would be different except for an already accounted negligible

probability; see above scenarios) and is negligible by the AEAD security of CreateFI and the negligible probability $p_{collIVFI}$ of FI IV collisions over all versions of the dir1/file1.

2 The forged FCH header of file dir1/file1 was taken from another file file2 that is a different version of dir1/file1. In this case, the same K_{FI} will be used to verify the forged FI block in file1 as was used to create the forged FI block in file2. However, this also leads to a forgery against AEAD security of CreateFI, except with negligible probability, thanks to the inclusion of the pair $(IV.Path, T.ACC, T.FCH)$ in 'associated data' input of FI encryption and the negligible probability $p_{collTFCH}$ of $T.FCH$ collisions over all versions of files over all files in the system.

Attack scenario: Modify/Create file data contents. The attacker modifies some of the data content of an encrypted file encdir1/encfile1 on StFS. When the file is opened with an Open call, and a Read call is made, no integrity error is detected, but a new content is read from the file, that has never been written to that file by a ScramFS function call.

How it is prevented in ScramFS: The authentication integrity security of the FC file content authenticated encryption algorithm encBlock implies that the success probability of such content forgery attacks is negligible, assuming the pseudorandomness of the underlying AES-256 block cipher.

Changing or creating a new ith content block without detection by the Read function implies (except for a previous version of the file that used the same $K_{FC(i)}$ also to authenticate some other content for the ith block – we deal with this case separately below) breaking the authentication security of the GCM scheme with respect to the file's ith block key $K_{FC(i)}$.

Namely, suppose we further modify the way we respond to the attacker's ScramFS calls from the scenario above by switching the FC content block ciphertexts to random strings and rejecting 'new' encrypted FC content blocks for the file when reading from the file (as in the 'ideal' ScramFS client, except we do not reject FC blocks $(C'.FC(i), T'.FC(i))$ that have appeared in the ith position of a previously created version of file1; we deal with this case separately below). Then the AEAD security of encBlock, and the negligible probability of $T.FCH$ collisions over all previous versions of file1 (see previous section), implies that the attacker can distinguish between the new game and the previous one with at most negligible distinguishing advantage p_4, and hence (since in the new game, the attacker succeeds in getting the forged FC content block accepted with zero probability) the FC content block forging probability is at most $p_1 + p_2 + p_3 + p_4 + p_5$, where

$$p_5 \leq \mathsf{Adv}^{AEAD}_{\mathsf{encBlock}}(N_f, N_{fc}, N_{fcpf}, q_{fv}, L_{blkmax}, T) + p_{collIVFC}, \tag{24}$$

which is negligible by (15) and (16).

To address the case of FC forgeries in which the 'new' ith FC block $(C'.FC(i), T'.FC(i))$ was taken from the ith position of a previous version of file1, we now further modify the way we respond to the attacker's ScramFS read calls from above by rejecting such 'new' ith blocks as well. If $T'.FC(i)$ in the forged block differs from the tag $T.FC(i)$ in the opened version of file1 (which is equal to the value $TList.FI[i]$ retrieved from FI in opening, or the file would have been rejected on opening by the FI forging scenario, see above) the forged block is rejected anyway before our modification due to the comparison of $TList.FI[i]$ with $T'.FC(i)$ in decBlock. Else, if $T'.FC(i) = T.FC(i)$ then we have a collision in generated $T.FC(i)$ tags among different versions of file1. Since now $T.FC(i)$'s are uniformly random for each block update, this collision event occurs with negligible probability, so the attacker can distinguish between the new game and the previous one with at most negligible distinguishing advantage p_6. Hence (since in the new game, the attacker succeeds in getting any forged FC content block accepted with zero probability) the

overall FC content block forging probability (including 'copied' blocks from old file versions) is at most $p_1 + p_2 + p_3 + p_4 + p_5 + p_6$, where

$$p_6 \leq p_{collFCT}, \tag{25}$$

which is negligible by (17). See scenario estimates below for examples.

4.2 Privacy Attacks

In this section, we summarize arguments for how common attack scenarios against the file system privacy are prevented in ScramFS, and estimate the success probabilities of those attack scenarios. In fact, essentially the same arguments as used for the integrity analysis also imply the privacy security, so below we only summarize the privacy-related aspects.

Attack Scenario: Distinguishing encrypted file/dir names from random strings: The attacker creates (on behalf of the client) a number of directories or files $\mathsf{dir}_i/\mathsf{name}_i$ for $i = 1, \ldots, q$ using CreateFile or Open ScramFS calls, and can distinguish between the corresponding encrypted file/dir names and character-encoded (via the underlying StFS.BinToStrEncode function) random bit strings, with non-negligible distinguishing advantage.

How it is prevented in ScramFS: The authenticated deterministic encryption (AED) security of the file/dir encryption algorithm $\mathsf{encName}'$ used in the ScramFS name encryption algorithm (including the pseudorandomness of the directory tree key derivation function – see previous section) implies that the distinguishing advantage of such attacks is negligible, assuming the pseudorandomness of the underlying AES-256 block cipher.

In particular, if the attacker calls ScramFS multiple times on the same file or directory (say $\mathsf{dir}_i/\mathsf{name}_i = \mathsf{dir}_j/\mathsf{name}_j$ for some i, j; for example opening a previously created file, the resulting filename ciphertext returned by ScramFS will be repeated (as a consequence of the deterministic $\mathsf{encName}'$ name encryption algorithm); the same behaviour will occur when the attacker interacts with the ScramFS 'ideal functionality', where the repeated file/dir name is indicated by the leaked file/dir access pattern DTAP.

Otherwise, suppose that dir_i or $\mathsf{dir}_i/\mathsf{name}_i$ is 'new' (is created for the first time), the returned ciphertext is indistinguishable from a random string as analysed in the 'Modification/Creation of a file/dir name' integrity scenario above. Moreover, due to the name padding in $\mathsf{encName}'$, the length of encrypted file/dir names depends on the encrypted file/dir name only via the leaked rounded name length $l = L.n_R(\mathsf{name}_i)$, which is the returned string length in the 'ideal functionality'.

Quantitatively, we may follow the same argument as in modified game discussed in the 'Modification/Creation of a file/dir name' integrity scenario, and estimate the attacker's distinguishing advantage against file/dir name privacy by p_1 bounded in (20).

Attack Scenario: Distinguishing encrypted file headers/footers from random strings: The attacker creates (on behalf of the client) a number of directories and files using CreateFile or Open (in write mode) ScramFS calls, and can distinguish between the corresponding encrypted file headers and random bit strings.

How it is prevented in ScramFS: The privacy (AEAD) security of the header and footer encryption algorithms and the use of non-repeating random nonces, implies that those blocks are indistinguishable from random (except with negligible probability). Furthermore, the authentication tags that link the headers together with the unique file name identifier also prevents active (chosen ciphertext) attacks, as already summarized in the integrity attack scenarios above. We also

observe that the lengths of header and footer ciphertext blocks is independent of the content of those blocks, as in the 'ideal functionality'.

Quantitatively, we may follow the same arguments as in modified games discussed in the 'Forge Header Block' and 'Forge Footer Block' integrity scenarios, and estimate the attacker's distinguishing advantage against file header/footer privacy by $p_1 + p_2 + p_3 + p_4$, where p_2, p_3, p_4 are bounded in (21), (22) and (23).

Distinguishing encrypted file content blocks from random strings: The attacker creates (on behalf of the client) a number of directories and files using CreateFile or Open (in write mode) ScramFS calls, and can distinguish between the corresponding encrypted file content blocks and random bit strings.

How it is prevented in ScramFS: The privacy (AEAD) security of the FC block encryption algorithm and the use of non-repeating random nonces, implies that those blocks are indistinguishable from random (except with negligible probability), under active (chosen ciphertext) attacks, as already described in the integrity attack scenarios above. We also observe that the length of encrypted FC blocks is the block size *Lblkov*, including the last padded FC block. This length reveals only the rounded file content length up to a multiple of *Lblkov*, as in the 'ideal functionality'.

Quantitatively, we may follow the same arguments as in modified game discussed in the 'Modify/Create file data contents' integrity scenario, and estimate the attacker's distinguishing advantage against privacy by $p_1 + p_2 + p_3 + p_4 + p_5 + p_6$, where p_5, p_6 are bounded in (24) and (25).

4.3 Summary of Quantitative Security Estimates for Typical Scenarios

In this section, we evaluate the quantitative security estimates from the previous section for two typical target application scenarios of ScramFS: a small business, and a data oriented user. For those two application scenarios, Table 1 summarizes the assumed file system usage parameters (corresponding to the 'online' attack parameters for an attacker interacting with the ScramFS client), while Table 2 summarizes the corresponding estimated security parameters for an offline classical attack run-time of $T = 2^{128}$ AES-256 evaluations or quantum attack run-time of $T \approx 2^{90}$ AES-256 evaluations. As explained in the previous Section, against classical computing attacks, we assume the best attack against the PRP security of AES-256 is exhaustive key-search with advantage estimated as in (2), whereas against quantum computer attacks, we assume the best attack against the PRP security of AES-256 is Grover's algorithm [4] applied to key search, with advantage estimated as in (3).

We note that even for the Data oriented scenario, the overall estimated upper bound on integrity attack success probability / privacy attack distinguishing advantage is negligible, namely at most 2^{-46}. This probability remains very small (less than $2^{-16} \approx 1/64,000$) even when considering an attack on any one of a very large set of 2^{30} (about 1 Billion) user file systems. This security level is estimated to hold even against quantum computing attacks.

4.4 'Post-quantum' Security

ScramFS was designed to provide long-term security even in the anticipated future 'post-quantum' scenario, in which large scale quantum computing machines are realized in practice. In view of this, we have included the quantitative estimates in the previous Section for security of ScramFS against quantum computing attacks. As remarked above, these estimates (Table 2) indicate that ScramFS maintains essentially the same level of 'online' security in the 'Post-quantum' scenario as

	Scenario:	Small Business	Data Oriented
		value	value
Parameter	**Meaning**		
N_{dir}	Total No. Dirs.	10^4	10^6
N_f	Total No. Files.	10^6	10^7
L_{name}	Max. file/dir name length (byte)	512	512
L_{blk}	ScramFS FC block length (byte)	10^5	10^5
L_{tot}	Total file data length (byte)	10^{11}	10^{13}
L_{fmax}	Maximum file data length (byte)	10^{11}	10^{11}
q_{fv}	No. accesses per file	10^3	10^4

Table 1. Assumed file system usage parameters for two typical scenarios.

	Scenario:	Small Business		Data Oriented	
		value (class.)	value (quant.)	value (class.)	value (quant.)
Parameter	**Meaning**				
p_1	File/Dir name forgery/dist. adv.	2^{-61}	2^{-60}	2^{-51}	2^{-51}
p_2	ACC hdr forging/dist. adv.	2^{-79}	2^{-61}	2^{-72}	2^{-54}
p_3	FCH hdr forging/dist. adv.	2^{-72}	2^{-56}	2^{-59}	2^{-52}
p_4	FI footer forging/dist. adv.	2^{-68}	2^{-56}	2^{-54}	2^{-52}
p_5	FC content forging/dist. adv.	2^{-62}	2^{-55}	2^{-46}	2^{-46}
p_6	FC tag collision probability	2^{-91}	2^{-91}	2^{-84}	2^{-84}
p_{tot}	Overall forgery/dist. adv.	2^{-60}	2^{-54}	2^{-46}	2^{-46}

Table 2. Estimated upper bounds on online attack success probabilities / privacy distinguishing advantage for two typical scenarios based on previous section, in both cases with offline classical attack run-time $T = 2^{128}$ AES-256 evaluations ('class.' values) or offline quantum attack run-time $T = 2^{90}$ quantum AES-256 evaluations ('quant.' values).

in the classical scenario, with the 'offline' security potentially lower than in the classical case but still excellent at around 2^{90} AES-256 evaluations (we remark that in this estimate, we make the reasonable assumption that the attacked ScramFS client system is running on a classical computer, and only the attack algorithm runs on a quantum computer). The latter level of offline security is the essentially the best security that can be hoped for in any block-cipher based on 256-bit keys, which becomes completely insecure at around 2^{128} block cipher operations due to Grover's quantum algorithm [4]. The use of AES-256 and the GCM authenticated encryption mode in ScramFS also conforms with the recommended best practice for cryptographic algorithms with long-term 'post-quantum' security according to the PQCRYPTO project, funded by the EU Commission of the European Communities [2].

4.5 Security Comparison with Other Encrypted File Systems

Security Comparison with EncFS. EncFS [15] is an Open-Source encrypted file system. In terms of functionality, at a high level, it uses a similar 'transparent encryption layer' approach to file system encryption as used in ScramFS, with a one-to-one correspondence between the plaintext directory tree structure and the encrypted directory tree structure. It also aims to achieve similar integrity and privacy security requirements as ScramFS. However, a security audit of EncFS version 1.7 was published in 2014 [14], which pointed out potential security issues in EncFS. We review these security issues below and explain why ScramFS does not suffer from similar issues. Unless otherwise specified, we refer to the vulnerable EncFS version 1.7 as 'EncFS' for short.

EncFS Issue: Same Key Used for Authentication and Encryption. In EncFS, the same key is used for both the Message Authentication Code (MAC) scheme used to provide file system integrity, as well as the encryption scheme used to provide confidentiality. As pointed out in the security review, this in general is considered as bad cryptographic practice, since it is known that even encryption and MAC schemes that are individually secure when used with independent random keys, may become insecure when they are both used with the same key.

Why ScramFS does not suffer from a similar issue. ScramFS uses carefully designed authenticated encryption schemes (SIV and GCM) to achieve integrity and privacy for the directory tree names and file headers and contents. These schemes have been specifically designed and shown [10,11] to be secure while simultaneously both authenticating and encrypting the plaintexts. In fact, the SIV authenticated encryption algorithm uses two independent keys, used for the internal authentication and encryption parts of the SIV algorithm. Similarly, the GCM authenticated encryption algorithm derives a pseudorandom 'authentication key' used for the internal authentication part of GCM, such that this pseudorandom authentication key is indistinguishable from a random key that is independent of the key-stream keys used in the CTR encryption part of GCM. Also, to ensure that all directories use keys that are indistinguishable from independent random keys, ScramFS uses pseudorandom key derivation functions to derive all directory tree keys from the master key.

EncFS Issue: Stream Cipher Used to Encrypt Last File Block. In EncFS, the last file content block is encrypted with a non-standard stream cipher. The security of this cipher is unknown.

Why ScramFS does not suffer from a similar issue. ScramFS uses the standard GCM authenticated encryption scheme to encrypt all file content blocks. This scheme was analysed and shown to be secure in [5,10].

EncFS Issue: Generating Block IV by XORing Block Number. In EncFS, given the File IV (an IV unique to a file), EncFS generates per-block IVs by XORing the File IV with the Block Number.

This leads to an IV-reuse attack due to the use of the stream cipher for the last content block. It may also lead to an IV-reuse attack if the same block in the same file is updated several times.

Why ScramFS does not suffer from a similar issue. In ScramFS, the per-block IV for the GCM authenticated encryption scheme is generated at random every time the block is updated and stored in the file. Due to the use of 128-bit IVs, the chance of IV-reuse occurs for two updates of the same block is negligible (see attack scenario analysis above). IV-reuse for updates to two different blocks within a file (or two blocks in different files) is not relevant to security since ScramFS uses a different pseudorandom key for each content block in each file, and file keys are chosen independently for each file and encrypted and authenticated in the file header.

EncFS Issue: File Holes are Not Authenticated. This allows an attacker to insert zero byte blocks ('holes') inside a file (or append zero blocks to the end of the file), without being detected when MAC headers are enabled.

Why ScramFS does not suffer from a similar issue. In ScramFS, all original file content and its length (including zero-byte blocks) are encrypted and authenticated. Adding such zero blocks to an encrypted file will cause the ScramFS file authentication integrity check to fail.

EncFS Issue: MACs Not Compared in Constant Time. In EncFS, MAC authentication tags are not compared in constant time. This allows an attacker with write access to the ciphertext to use a timing attack to compute the MAC of arbitrary values.

Why ScramFS does not suffer from a similar issue. In ScramFS implementations, MAC authentication tags are compared with a constant-time routine.

EncFS Issue: 64-bit MACs. EncFS uses 64-bit MACs. This may not be long enough for sufficient security.

Why ScramFS does not suffer from a similar issue. The ScramFS specification uses 128-bit MAC authentication tags, for both SIV file/dir name authentication as well as GCM file header and content authentication. Assuming the pseudorandomness of AES-256, the probability of forgery is negligible for typical parameters (see security analysis above).

EncFS Issue: Editing Configuration File Disables MACs. The MAC authentication security in EncFS can be disabled by an attacker that modifies an unauthenticated configuration file on the file system.

Why ScramFS does not suffer from a similar issue. ScramFS does not rely on unauthenticated configuration files to enable or disable integrity checking.

References

1. D. Boneh and V. Shoup. *A Graduate Course in Applied Cryptography.* 2015. Available at `https://crypto.stanford.edu/ dabo/cryptobook/`.
2. D. Augot et al. Initial recommendations of long-term secure post-quantum systems, 2015. Available at `https://pqcrypto.eu.org/docs/initial-recommendations.pdf`.
3. O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *J. ACM*, 33(4):792–807, 1986.
4. L. Grover. A fast quantum mechanical algorithm for database search. In *Proc. of STOC*. ACM, 1996.
5. T. Iwata, K. Ohashi, and K. Minematsu. Breaking and repairing GCM security proofs. In *Proc. of CRYPTO 2012*, LNCS. Springer, 2012. Available at `https://eprint.iacr.org/2012/438.pdf`.
6. J. Katz and Y. Lindell. *Introduction to Modern Cryptography.* Chapman and Hall, 2014.
7. Morris Dworkin. NIST Special Publication 800-38A, 2001 Edition. Recommendation for Block Cipher Modes of Operation. Available at http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf.
8. Morris Dworkin. NIST Special Publication 800-38D, 2007 Edition. Galois/Counter Mode (GCM) and GMAC. Available at http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf.
9. NIST. FIPS 197, 2001 Edition. Specification for the ADVANCED ENCRYPTION STANDARD (AES). Available at http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

10. Y. Niwa, K. Ohashi, K. Minematsu, and T. Iwata. GCM security bounds reconsidered. In *Proc. of FSE 2015*, LNCS. Springer, 2015. Available at `https://eprint.iacr.org/2015/214.pdf`.
11. P. Rogaway and T. Shrimpton. Deterministic Authenticated-Encryption. IACR Cryptology ePrint Archive, report 2006/221, Aug. 20, 2007. Available at https://eprint.iacr.org/2006/221.pdf.
12. R. Steinfeld. ScramFS: a simple file system symmetric encryption layer: Design requirements and specifications. Scram Software Document.
13. P. Rogaway. Authenticated-encryption with associated-data. In *Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS 2002, Washington, DC, USA, November 18-22, 2002*, pages 98–107, 2002. Full version available at http://web.cs.ucdavis.edu/ rogaway/papers/ad.pdf.
14. T. Hornby. EncFS security audit, 2014. Available at `https://defuse.ca/audits/encfs.htm`.
15. V. Gough. EncFS – an encrypted filesystem. Available at `https://vgough.github.io/encfs/`.