Security Audit of `gocryptfs` v1.2

Focus: Main-Feature Cryptography Design

Taylor Hornby

taylor@defuse.ca

Defuse Security

February 25, 2017

Contents

Abstract

This report documents a two-day audit of the `gocryptfs` encrypted filesystem. Unlike full-disk encryption systems, `gocryptfs` encrypts files individually using chunked AES-GCM (Galois Counter Mode) and encrypts filenames with AES-EME (ECB-Mix-ECB). Our audit focused on the crypgography design of `gocryptfs`'s main file encryption features; it excluded its dependencies and its more complicated "Reverse Mode" feature that uses deterministic AES-SIV (Synthetic Initialization Vector) encryption. We did not look at the implementation code except when it was necessary to understand some aspect of the design.

We found that `gocryptfs` provides excellent confidentiality against a passive adversary, i.e. one that does not tamper with the encrypted files. On the other hand, we found that `gocryptfs` provides no security at all against an active adversary who can modify the ciphertexts while having read access to any subdirectory of the mounted filesystem. Against a less-powerful active adversary who can modify the ciphertexts but has *no* access to the mounted filesystem, `gocryptfs` keeps file contents secret and provides imperfect integrity protection. In at least one case, imperfections in the integrity protections lead to a break of *confidentiality*. It is possible that the integrity imperfections lead to further confidentiality breaks depending on which applications are using the filesystem.

We believe the reason these vulnerabilities exist is because `gocryptfs` doesn't have a clearly spelled-out threat model. Some of the attacks seem hard to avoid given `gocryptfs`'s performance goals and may have been introduced "by design" to meet these goals. We suggest writing down an explicit threat model and updating the website to better communicate the security guarantees that `gocryptfs` provides. This way, users are less likely to rely on it in ways which would make them vulnerable.

1 Introduction

This report describes the findings of a two-day security audit of the `gocryptfs` encrypted filesystem. From the project's web page [5],

> *gocryptfs uses file-based encryption that is implemented as a mountable FUSE filesystem. Each file in gocryptfs is stored [in] one corresponding encrypted file on the hard disk...*
>
> *The encrypted files can be stored in any folder on your hard disk, a USB stick or even inside the Dropbox folder. One advantage of file-based encryption as opposed to disk encryption is that encrypted files can be synchronised efficiently using standard tools like Dropbox or rsync. Also, the size of the encrypted filesystem is dynamic and only limited by the available disk space.*

We audited version 1.2 of `gocryptfs`, specifically Git revision `9b57384`.

1.1 Audit Scope

Our audit was very short (just two days), so it focused exclusively on `gocryptfs`'s cryptography design for its main use case. We looked at the implementation, but only when it was necessary to understand the design. The following potential sources of vulnerability were *not* examined.

- Most of the `gocryptfs` source code.
- The implementations of crypto the primitives Scrypt, GCM, EME, and AES.
- The proofs of GCM's and EME's security properties.
- The implementation of the `crypto/rand` cryptographically-secure random number generator that `gocryptfs` relies on for generating random keys, salts, and IVs.
- Non-cryptographic bugs in `gocryptfs`, like remote code execution bugs and bugs in the filesystem implementation that could lead to accidental too-open access permissions.
- `gocryptfs`'s "Reverse Mode" as well as normal operation with AES-SIV instead of AES-GCM.

- Whether or not `gocryptfs`'s in-memory keys can be leaked to the system's swap file (there's no code to lock keys in memory, so they probably are).
- Side-channel attacks.

The most notable omission is that we did not look into the "Reverse Mode" design or implementation. In "Reverse Mode", `gocryptfs` gives you an encrypted view of a directory of plaintext files. In this mode, `gocryptfs` encrypts the files deterministically using AES-SIV so that re-mounting the same directory results in the same ciphertexts. This feature is important for making efficient incremental backups possible. Deterministic encryption is difficult to get right, so we strongly recommend a future audit that focuses exclusively on the AES-SIV features.

1.2 Threat Model

For the purposes of this audit, we've defined three stereotypical kinds of adversary with the aim of capturing a variety real-world attack scenarios. *Eve* represents an adversary who has the ability to write to the mounted filesystem and has read-only access to the `gocryptfs` ciphertext as it changes over time. *Mallory* is an adversary who has full read-write access to the ciphertext directory as well as read-write access to part of the mounted filesystem. *Dropbox* is in roughly between Eve and Mallory, with full read-write access to the ciphertext but no access at all to the mounted filesystem.

	Ciphertext	**Mounted filesystem**
Eve	Read-only	Write-only
Dropbox	Read-write	No access
Mallory	Read-write	Read-write to some directories

For example, if you use `gocryptfs` to encrypt files on a USB drive, and then you lose the drive and someone finds it, you need to be secure against Eve. We've given Eve write-only access to the mounted filesystem to account for chosen plaintext attacks, e.g. Eve can social-engineer the victim into placing a file of Eve's choice onto the `gocryptfs` filesystem before gaining access to the ciphertext. If you're uploading `gocryptfs`-encrypted files to the cloud, then you need to be secure against Dropbox, since the cloud storage provider can both see and modify the ciphertext, but doesn't have access to any part of the mounted filesystem. Mallory corresponds to giving an

untrusted user full access to the ciphertext directory as well as to some subdirectories of the mounted filesystem; it's the most powerful kind of adversary `gocryptfs` could reasonably be expected to be secure against. Taking social-engineering chosen-plaintext attacks into account, an actual cloud storage provider's capabilities may be closer to Mallory than to Dropbox.

The vulnerabilities we discovered are classified according to which of the three adversaries can exploit the vulnerability and what they can gain by doing so.

2 Findings

`gocryptfs` doesn't have a documented threat model, so it's unclear what counts as a vulnerability and what doesn't. Not having a threat model made the audit more difficult, since we had to guess what security guarantees the average user would expect `gocryptfs` to provide after they've read the project's website. We expect some of our findings (like some of the file-level ciphertext malleability attacks) to turn out to be "known issues" which simply aren't documented and aren't communicated clearly to the users. In Section 4.1 we recommend creating a threat model similar to the one we gave above and documenting which security properties are expected vs. not expected to be provided.

In the following sections, we describe the vulnerabilities that we found. We classify each vulnerability according to which of our threat model adversaries (Eve, Dropbox, and Mallory) can exploit it and summarize the consequences of their exploitation.

2.1 File-Level Ciphertext Malleability

	Exploitable	**Consequences**
Mallory	Yes.	Complete break of file integrity and confidentiality.
Dropbox	Yes.	Files can be fully or partially restored from earlier versions, duplicated, made to have the same contents as another file, deleted, truncated, and moved. These integrity problems could turn into *confidentiality* problems depending on the applications that use the filesystem.
Eve	No.	-

In `gocryptfs`, integrity protection works as follows. In the header of each ciphertext file, there is a value called the *file ID*. Files are encrypted using AES-GCM

in chunks of 4096 bytes. To prevent chunks from being re-ordered within a file or replaced by chunks from other files, the file ID and chunk number are included in the additional authenticated data input to GCM.

The problem is that the integrity of the file contents is bound just to the file ID and *not* to the file name and/or file path. Exchanging the (encrypted) names of two ciphertext files exchanges their plaintext contents.

Mallory can decrypt any ciphertext file simply by copying the ciphertext into a ciphertext directory corresponding a directory he has access to in the mounted filesystem. `gocryptfs` has no way to know the file was copied from somewhere else, so the original plaintext file shows up in the subdirectory of the mounted filesystem that Mallory has access to, and now Mallory has access to the plaintext. This is demonstrated in Proof of Concept (PoC) 1 in Appendix A.

Mallory can also replace any plaintext file with contents of his choice. All he has to do is reverse the process above: write the contents to a file in a directory he has access to, and then copy the ciphertext file over top of the target ciphertext file that he would like to modify. This is demonstrated in PoC 2 in Appendix A.

Note that the file permissions of the ciphertext are the same as the plaintext, so this is assuming Mallory has the ability to see and modify *all* of the ciphertext, yet for some reason can only see and modify *some* of the plaintext. This is certainly possible: for example imagine the `gocryptfs` ciphertext is kept synchronized with a cloud storage provider, and the user has a cronjob that regularly publishes a non-secret file (say, their favorite text editor's configuration) to their website. The cloud storage provider can replace the cron-uploaded file with any ciphertext file, wait for the cronjob to run, and then download the plaintext.

Having read-write access to the ciphertext without access to the mounted filesystem, it's possible to swap the contents of two files (see PoC 3), restore files to earlier versions, restore *certain chunks* of files from earlier versions, duplicate files, delete files, and truncate files to any multiple of 4096 bytes (PoC 4). Fixing some of these issues would be in conflict with `gocryptfs`'s performance goals. It's reasonable to allow some degree of ciphertext malleability in favor of performance, but it's risky given that depending on what kinds of applications are using the filesystem, these weaknesses might be used to steal plaintext too. For example, the adversary might know the user is about to send them a file and swap it for a different one just before it gets sent.

2.2 File ID Poisoning

	Exploitable	Consequences
Mallory	Not necessary.	Because of Issue 2.1, Mallory doesn't need to exploit this vulnerability to completely break file integrity.
Dropbox	Yes.	Dropbox can create special ciphertext files whose future-written chunks can be swapped for the chunks in other files without detection. Depending on the applications using the filesystem, these integrity problems could turn into *confidentiality* problems.
Eve	No.	-

Another effect of the file ID not being tied to the file name or file path is that an adversary with access to the ciphertext can cause *two* files to have the same file ID, and then any chunks that the user writes to those files can be swapped. For example, the adversary can notice when the user creates a new empty file and immediately "poison" it to have the same file ID as some other file. Now, all the chunks the user writes to the new file can be exchanged for chunks in the other file. This is demonstrated in PoC 5.

When this is done, chunks can only be swapped with other chunks that were written to the same position (same offset into the file) because the chunk number is included in the authenticated data alongside the file ID.

In the most extreme case, the adversary could poison *all* files to have the same file ID, so that chunks can be swapped between any two files. However, this probably won't be possible in practice because changing the file ID of a nonempty file breaks the integrity check of all its chunks—the adversary has set the file ID only when the file is empty, or it'll be detected.

2.3 Directory IV Poisoning

	Exploitable	Consequences
Mallory	Yes.	Complete break of filename integrity and confidentiality.
Dropbox	Yes.	Over time, and at some risk of being detected, Dropbox can determine whether files in different directories have the same name.
Eve	No.	-

Filenames are encrypted using EME mode[1] with an Initialization Vector (IV) from the file `gocryptfs.diriv` in the ciphertext directory. The IV is not authenticated, so similar to the file ID poisoning attack, an active adversary can force two different directories to use the same IV.

In the case of Mallory, this completely breaks filename confidentiality. Mallory can just copy the `gocryptfs.diriv` file from the victim's ciphertext directory into a ciphertext directory she controls, then create files in that directory with the same encrypted names, and then do a file listing on the corresponding plaintext directory in the mounted filesystem. The listing will show the decrypted names of all the files that were in the victim's ciphertext directory. This is demonstrated in PoC 6. Mallory can also create files with arbitrary names by setting the two directory IVs to the same thing, creating a file with the desired name in a subdirectory of the mountpoint that she has access to and then creating a file with the same encrypted name in the victim's ciphertext directory.

To exploit this, attackers without access to any part of the mounted filesystem can only poison two directories to have the same directory IV and then watch to observe if any of the names in one directory ever match any of the names in the other. This is demonstrated in PoC 7. Changing the directory IV will break the decryption of the filenames of all the files that currently exist in the directory, so carrying out this attack in practice will probably involve some risk of being detected.

2.4 Same Key Used for Both GCM and EME Modes

	Exploitable	**Consequences**
Mallory	Unknown, likely.	Potential confidentiality/integrity problems.
Dropbox	Unknown, less likely.	Potential confidentiality/integrity problems.
Eve	Unknown, probably not.	Potential confidentiality problems.

The same 32-byte master key gets reused for AES-GCM and AES-EME modes. There are security proofs for GCM and EME *on their own* but reusing the same key for both invalidates their security proofs. The reason is that, especially since they are both using the same block cipher, there could be some sort of interaction between them where properties of one of them makes it possible to break the other

[1]For those unfamiliar, EME takes a block cipher that operates on small blocks and builds from it a block cipher that operates on much larger blocks.

one. `gocryptfs` should use two independent keys for GCM and EME. These keys can be derived from the master key using something like HKDF or by simply increasing the master key's length to 64 bytes and using first half for GCM and the second half for EME.

2.5 No Integrity Protection for File Permissions

	Exploitable	**Consequences**
Mallory	Depends.	Complicated; see description.
Dropbox	Depends.	Complicated; see description.
Eve	No.	-

`gocryptfs` passes file permissions through to the ciphertext directory and they are not integrity protected. This means that if an adversary can modify the file permissions of the ciphertext, they can modify the file permissions inside the mounted filesystem. This could be exploitable in some circumstances.

For example, suppose a user creates a system backup using `gocryptfs` and then archives it with `tar`, telling `tar` to preserve the file permissions, and then uploads the backup to the cloud. Say the system has a read-protected secret TLS key at `/etc/ssl/private-key`. The cloud provider could change the `tar` archive to make that file globally-readable. When the user restores from the backup, their TLS key is vulnerable to being stolen by untrusted users of the system.

Thankfully, the `suid` bit must be explicitly enabled in `gocryptfs` with a command-line option to the mount command, so in the default configuration Mallory can't take advantage of this to run code as `root`.

`gocryptfs` should either document this weakness or add some sort of integrity protection of the file permission bits.

2.6 Pushing the Limits of GCM

	Exploitable	**Consequences**
Mallory	Probably not.	Potential recovery of plaintext and/or the internal authentication key used by GCM.
Dropbox	Probably not.	Potential recovery of plaintext and/or the internal authentication key used by GCM.
Eve	Probably not.	Potential recovery of plaintext and/or the internal authentication key used by GCM.

According to NIST's recommendations on GCM [6],

> *The total number of invocations of the authenticated encryption function shall not exceed 2^{32}, including all IV lengths and all instances of the authenticated encryption function with the given key.*

In other words, using the same key and random initialization vectors (either 96 bit or 128 bit), it's unsafe to call the encryption function more than 2^{32} times. In the context of `gocryptfs` this would mean that no more than 2^{32} *chunks* can ever be created safely, since each for each written chunk a new random IV is generated. Chunks are 4096 bytes, so that would mean the maximum amount of data you can safely write—according to this guidance—to a single `gocryptfs` filesystem over all of its lifetime is 16 TiB.

However, the paragraph quoted above assumes up to 2^{32} blocks are passed to the encryption function in each call, and `gocryptfs` only ever passes in chunks of 2^8 blocks (4096 bytes). So it is actually safe to write more than hundreds of terabytes of data to a `gocryptfs` repository in its lifetime [2].

It's important to understand that this not a limit on the amount of data you can keep in a single `gocryptfs` filesystem at one time, it's a limit on the *amount of data you can write to the filesystem in its entire lifetime*. Users who expect to be writing more than, say, a petabyte of data to a `gocryptfs` repository in its lifetime should repeat the calculations in [2] to make sure they don't hit these limits.

It would be nice for `gocryptfs` to switch to a better AEAD construction once one becomes available [3].

3 Good Things

As well as finding flaws, a security audit should point out things that were well done so that other projects can take note and adopt the practices. Here are some things that `gocryptfs` got right.

3.1 Clear Protocol Design Documentation

The crypto design documentation is pretty clear, and having it available helped this audit cover a lot more ground than it would have otherwise. The documentation could be improved slightly by making note of how `gocryptfs` generates random keys, IVs, and file IDs, and by noting what the default Scrypt parameters are.

3.2 No Fallback to Old On-Disk Formats

Older versions of `gocryptfs` used different on-disk formats. The author made a great design decision by not trying to support mounting older versions by falling back to the older (weaker) crypto designs. Falling back to older protocols is a common source of vulnerability, and `gocryptfs` avoids it.

4 Recommendations and Future Work

In the next sections we give some recommendations for making `gocryptfs` more secure and say what else needs to be audited.

4.1 Threat Model

Most of the vulnerabilities we found could have been spotted in the process of writing down a threat model for `gocryptfs`. We recommend creating a simple threat model by listing a few different kinds of adversary that the users might want to defend themselves against[2] and laying out exactly what security properties `gocryptfs` should provide against the adversaries. This is worth documenting, because the security goals aren't likely to change, so the document won't "rot" and it will be extremely useful to future security auditors as well as users to who want to understand what precise level of protection `gocryptfs` actually provides.

4.2 In-depth Audit of the Implementation

As mentioned, we only looked at the implementation code when it was necessary to understand the design. Obviously, it would be valuable to take a careful look at the implementation to make sure it matches the design.

4.3 Audit the EME Implementation

For GCM, `gocryptfs` uses either OpenSSL or the `crypto/cipher` package. These are widely used, so although they should be audited independently, we feel like they're less likely to be incorrect. On the other hand, the implementation of EME that `gocryptfs` uses [4] is not widely used (it appears to have been written specifically for `gocryptfs`) so we strongly recommend having it reviewed. The EME implementation's test suite

[2] Feel free to copy our definitions of Mallory, Dropbox, and Eve.

does compare it against "official" test vectors, and that makes us more confident in its correctness, but test vectors are no substitute for a careful analysis.

4.4 Audit Reverse Mode and AES-SIV

We did not look at the deterministic-encryption "Reverse Mode" feature nor did we look for problems when `gocryptfs` is operating in normal mode with AES-SIV. It would take us an extra one or two days to properly understand the nonce-reuse-resistance properties of AES-SIV and then determine whether the `gocryptfs` features use it safely.

4.5 Audit for Non-Cryptographic Vulnerabilities

We did not look for non-cryptography-related bugs. There are other important aspects to `gocryptfs`'s security beyond cryptography. For example, it's important that file permissions are handled correctly. It's also important that there are no remote-code-execution-style bugs, especially if the user chooses to run `gocryptfs` as `root`. We didn't spend any time looking for these problems, so looking for them is left to future audits.

5 Conclusion

We found that `gocryptfs`'s security ranges from great to extremely poor depending on the setting in which it is used. With the current design, users MUST ensure that no attacker can modify the ciphertext *and* read from some part of the mounted filesystem, otherwise there will be a catastrophic security failure. Users must also be aware that `gocryptfs` provides imperfect integrity protections against less-powerful kinds of adversaries, and that those imperfections might lead to confidentiality leaks when certain applications are run on top of a `gocryptfs` filesystem. For the typical scenario of an adversary gaining access to a static copy of the ciphertext, `gocryptfs` provides good confidentiality protection as long as it's used with a strong passphrase.

Users must be made aware of the real guarantees that `gocryptfs` provides. They must make sure they're *only* relying on those guarantees—not on any other security properties they intuitively expect `gocryptfs` to provide but actually aren't provided.

6 Acknowledgements

I would like to thank 23andMe [1] for funding this audit. Without their support this audit would not have been possible. Thanks also to the `gocryptfs` author Jakob Unterwurzacher for correcting an error in an earlier version of this report.

References

[1] 23andMe: DNA and genetic testing & analysis.
https://www.23andme.com/.

[2] Github comment on issue 17.
https://github.com/rfjakob/gocryptfs/issues/17#issuecomment-169020984.

[3] CAESAR submissions.
https://competitions.cr.yp.to/caesar-submissions.html.

[4] EME (encrypt-mix-encrypt) wide-block encryption for Go.
https://github.com/rfjakob/eme.

[5] gocryptfs - simple. secure. fast.
https://nuetzlich.net/gocryptfs/.

[6] Morris J Dworkin. SP 800-38D. Recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC. 2007.

Appendix A Proofs-of-Concept

PoC 1: Confidentiality breakage by Mallory

Alice creates a directory containing a file with sensitive contents:

```
$ mkdir gocryptfs-mountpoint/alice
$ echo "Super secret file contents" > gocryptfs-mountpoint/
    ↪ alice/secret-file.txt
```

Mallory locates Alice's directory's ciphertext:

```
$ ls gocryptfs-ciphertext/
gocryptfs.conf  gocryptfs.diriv  jJlZrtxzy-1QAjZ8spB3rw==
```

Mallory copies Alice's directory's ciphertext to a directory containing the ciphertext of a directory Mallory has access to:

```
$ mkdir gocryptfs-mountpoint/mallory
$ cp -fR gocryptfs-ciphertext/jJlZrtxzy-1QAjZ8spB3rw\=\=/*
    ↪ gocryptfs-ciphertext/U1a3Wv6ApxRVo-8f0BH4cw\=\=/
```

Mallory learns the filename and its contents:

```
$ ls gocryptfs-mountpoint/mallory/
secret-file.txt
$ cat gocryptfs-mountpoint/mallory/secret-file.txt
Super secret file contents
```

PoC 2: Integrity breakage by Mallory

Alice creates a file she believes is integrity-protected:

```
$ mkdir gocryptfs-mountpoint/alice
$ echo "I don't want this file to be changed." > gocryptfs-
    ↪ mountpoint/alice/integrity-protected-file.txt
$ cat gocryptfs-mountpoint/alice/integrity-protected-file.txt
I don't want this file to be changed.
```

Mallory locates Alice's directory's ciphertext:

```
$ ls gocryptfs-ciphertext/
gocryptfs.conf  gocryptfs.diriv  jJlZrtxzy-1QAjZ8spB3rw==
```

Mallory copies Alice's directory's ciphertext to a directory containing the ciphertext of a directory Mallory has access to:

```
$ mkdir gocryptfs-mountpoint/mallory
$ cp -fR gocryptfs-ciphertext/jJlZrtxzy-1QAjZ8spB3rw\=\=/*
    ↪ gocryptfs-ciphertext/U1a3Wv6ApxRVo-8f0BH4cw\=\=/
```

Mallory learns the filename and its contents:

```
$ ls gocryptfs-mountpoint/mallory/
secret-file.txt
$ cat gocryptfs-mountpoint/mallory/secret-file.txt
Super secret file contents
```

PoC 3: File Contents Swapping by Dropbox

Alice creates two files:

```
$ mkdir gocryptfs-mountpoint/alice
$ echo "FILE 1" > gocryptfs-mountpoint/alice/FILE1.txt
$ echo "FILE 2" > gocryptfs-mountpoint/alice/FILE2.txt
```

Mallory swaps their ciphertexts:

```
$ ls gocryptfs-ciphertext/
gocryptfs.conf            gocryptfs.diriv           jJlZrtxzy
   ↪ -1QAjZ8spB3rw==/
$ ls gocryptfs-ciphertext/jJlZrtxzy-1QAjZ8spB3rw\=\=/
gocryptfs.diriv  LyhDOFBnRCQH15k9ibNAGA==  rPOE4n57odzoY1-XL-
   ↪ EsUg==
$ mv gocryptfs-ciphertext/jJlZrtxzy-1QAjZ8spB3rw\=\=/
   ↪ rPOE4n57odzoY1-XL-EsUg\=\= /tmp/for-swap
$ mv gocryptfs-ciphertext/jJlZrtxzy-1QAjZ8spB3rw\=\=/
   ↪ LyhDOFBnRCQH15k9ibNAGA\=\= gocryptfs-ciphertext/
   ↪ jJlZrtxzy-1QAjZ8spB3rw\=\=/rPOE4n57odzoY1-XL-EsUg\=\=
$ mv /tmp/for-swap gocryptfs-ciphertext/jJlZrtxzy-1
   ↪ QAjZ8spB3rw\=\=/LyhDOFBnRCQH15k9ibNAGA\=\=
```

Alice notices that the plaintexts have been swapped:

```
$ cat gocryptfs-mountpoint/alice/FILE1.txt
FILE 2
$ cat gocryptfs-mountpoint/alice/FILE2.txt
FILE 1
```

PoC 4: File Truncating by Dropbox

Alice creates a two-chunk file:

```
$ mkdir gocryptfs-mountpoint/alice
$ perl -e 'print "A"x4096 . "B"x4096' > gocryptfs-mountpoint/
   ↪ alice/twochunks.txt
$ wc -c gocryptfs-mountpoint/alice/twochunks.txt
8192 gocryptfs-mountpoint/alice/twochunks.txt
$ tail -c 20 gocryptfs-mountpoint/alice/twochunks.txt
BBBBBBBBBBBBBBBBBBBB
```

Dropbox chops off everything but the header and first chunk in the ciphertext:

```
$ truncate --size=4146 gocryptfs-ciphertext/jJlZrtxzy-1
   ↪ QAjZ8spB3rw\=\=/iaJxxqvcukwm3puG5CVNgQ\=\=
```

Alice finds that the last 4096 bytes of the file are missing:

```
$ wc -c gocryptfs-mountpoint/alice/twochunks.txt
4096 gocryptfs-mountpoint/alice/twochunks.txt
$ tail -c 20 gocryptfs-mountpoint/alice/twochunks.txt
AAAAAAAAAAAAAAAAAAAA
```

PoC 5: File ID Poisoning by Dropbox

Alice creates a two-chunk file:

```
$ mkdir gocryptfs-mountpoint/alice
$ perl -e 'print "A"x4096 . "B"x4096' > gocryptfs-mountpoint/
   ↪ alice/AB.txt
```

Dropbox finds out where that file's ciphertext is:

```
$ ls gocryptfs-ciphertext/jJlZrtxzy-1QAjZ8spB3rw\=\=/
gocryptfs.diriv  zJbsDcElB33TRWgx1int-Q==
```

Alice creates an empty file, planning to write to it later:

```
$ touch gocryptfs-mountpoint/alice/CD.txt
```

Dropbox poisons the file ID of the new file to be the same as the old one:

```
$ head -c 18 gocryptfs-ciphertext/jJlZrtxzy-1QAjZ8spB3rw\=\=/
   ↪ zJbsDcElB33TRWgx1int-Q\=\= > gocryptfs-ciphertext/
   ↪ jJlZrtxzy-1QAjZ8spB3rw\=\=/zq0DsbuVEJT8xCejmnYuSQ\=\=
```

Alice appends two different blocks to the new file:

```
$ perl -e 'print "C"x4096 . "D"x4096' >> gocryptfs-mountpoint
   ↪ /alice/CD.txt
```

Dropbox checks that the file IDs are stil the same:

```
$ xxd -l 18 gocryptfs-ciphertext/jJlZrtxzy-1QAjZ8spB3rw\=\=/
   ↪ zJbsDcElB33TRWgx1int-Q\=\=
00000000: 0002 a856 75e1 eeed 1191 6c54 6a3e 19bf  ...Vu.....
   ↪ lTj>..
00000010: c8ae                                     ..
$ xxd -l 18 gocryptfs-ciphertext/jJlZrtxzy-1QAjZ8spB3rw\=\=/
   ↪ zq0DsbuVEJT8xCejmnYuSQ\=\=
00000000: 0002 a856 75e1 eeed 1191 6c54 6a3e 19bf  ...Vu.....
   ↪ lTj>..
```

```
00000010: c8ae                                     ..
```

Dropbox transplants the second block of the new file into the old one:

```
$ tail -c 4128 gocryptfs-ciphertext/jJlZrtxzy-1QAjZ8spB3rw
   ↪ \=\=/zq0DsbuVEJT8xCejmnYuSQ\=\= | dd of=gocryptfs-
   ↪ ciphertext/jJlZrtxzy-1QAjZ8spB3rw\=\=/
   ↪ zJbsDcElB33TRWgx1int-Q\=\= bs=1 seek=4146 conv=notrunc
4128+0 records in
4128+0 records out
4128 bytes (4.1 kB, 4.0 KiB) copied, 0.00481959 s, 857 kB/s
```

Alice finds that the second block of the old file has been changed to the contents of the second block of the new file:

```
$ head -n 20 gocryptfs-mountpoint/alice/AB.txt
AAAAAAAAAAAAAAAAAAAA
$ tail -c 20 gocryptfs-mountpoint/alice/AB.txt
DDDDDDDDDDDDDDDDDDDD
```

PoC 6: Directory IV Poisoning by Mallory

Alice creates a directory and puts a file with a secret filename inside:

```
$ mkdir gocryptfs-mountpoint/alice
$ touch gocryptfs-mountpoint/alice/super-secret-filename.txt
```

Mallory finds Alice's encrypted filename:

```
$ mkdir gocryptfs-mountpoint/mallory
$ rmdir gocryptfs-mountpoint/mallory/
$ ls gocryptfs-ciphertext/
gocryptfs.conf  gocryptfs.diriv  jJlZrtxzy-1QAjZ8spB3rw==
$ ls gocryptfs-ciphertext/jJlZrtxzy-1QAjZ8spB3rw\=\=/
gocryptfs.diriv  us7db7UHSvqRsC2l9CVIyrjDXybEonD39J2Rzqc0rYU=
```

Mallory creates a directory with the same directory IV as Alice's:

```
$ mkdir gocryptfs-mountpoint/mallory
$ cp -f gocryptfs-ciphertext/jJlZrtxzy-1QAjZ8spB3rw\=\=/
   ↪ gocryptfs.diriv gocryptfs-ciphertext/U1a3Wv6ApxRVo-8
   ↪ f0BH4cw\=\=/gocryptfs.diriv
```

Mallory creates an empty ciphertext file with the encrypted filename, and discovers the decrypted name:

```
$ touch gocryptfs-ciphertext/U1a3Wv6ApxRVo-8f0BH4cw\=\=/
   ↪ us7db7UHSvqRsC2l9CVIyrjDXybEonD39J2Rzqc0rYU=
$ ls gocryptfs-mountpoint/mallory/
super-secret-filename.txt
```

PoC 7: Directory IV Poisoning by Dropbox

Alice creates two different directories, and puts a file in the first one:

```
$ mkdir gocryptfs-mountpoint/alice1
$ mkdir gocryptfs-mountpoint/alice2
$ touch gocryptfs-mountpoint/alice1/secret-filename.txt
```

Dropbox forces the second directory to have the same IV as the first:

```
$ cp -f gocryptfs-ciphertext/KVp0vsfPuiyI4OQyHTxdWg\=\=/
   ↪ gocryptfs.diriv gocryptfs-ciphertext/
   ↪ pO2O9zGKjXAoCppKUg_iZQ\=\=/gocryptfs.diriv
```

Alice creates a file in the second directory with the same name:

```
$ touch gocryptfs-mountpoint/alice2/secret-filename.txt
```

Dropbox sees that both files have the same name:

```
$ ls gocryptfs-ciphertext/KVp0vsfPuiyI4OQyHTxdWg\=\=/
DNMZOT9ZzR5ey6K1JBhmkRvlTu5Qxt3qo4XqzOHdOjg=  gocryptfs.diriv
$ ls gocryptfs-ciphertext/pO2O9zGKjXAoCppKUg_iZQ\=\=/
DNMZOT9ZzR5ey6K1JBhmkRvlTu5Qxt3qo4XqzOHdOjg=  gocryptfs.diriv
```