3-2-2019

Frameup: An Incriminatory Attack on Storj: A Peer to Peer Blockchain Enabled Distributed Storage System

Xiaolu Zhang
University of Texas at San Antonio

Justin Grannis
University of New Haven

Ibrahim Baggili
University of New Haven, ibaggili@newhaven.edu

Nicole Lang Beebe
University of Texas at San Antonio

Publisher Citation

Zhang, X., Grannis, J., Baggili, I., & Beebe, N. L. (2019). Frameup: An incriminatory attack on storj: A peer to peer blockchain enabled distributed storage system. Digital Investigation Volume 29, June 2019, pp. 28-42. doi:10.1016/j.diin.2019.02.003

Comments

This is the authors' accepted version of the article published in *Digital Investigation*. The version of record can be found at http://dx.doi.org/10.1016/j.diin.2019.02.003

Dr. Baggili was appointed to the University of New Haven's Elder Family Endowed Chair in 2015.

Frameup: An Incriminatory Attack on Storj: A Peer to Peer Blockchain Enabled Distributed Storage System

Xiaolu Zhang[a,*], Justin Grannis[b], Ibrahim Baggili[b], Nicole Lang Beebe[a]

[a] *Information Systems & Cyber Security Department, The University of Texas at San Antonio, One UTSA Circle, San Antonio, TX, 78249*
[b] *Cyber Forensics Research & Education Group, Tagliatela College of Engineering, ECECS, University of New Haven, 300 Boston Post Rd., West Haven, CT, 06516*

Abstract

In this work we present a primary account of *frameup*, an incriminatory attack made possible because of existing implementations in distributed peer to peer storage. The frameup attack shows that an adversary has the ability to store unencrypted data on the hard drives of people renting out their hard drive space. This is important to forensic examiners as it opens the door for possibly framing an innocent victim. Our work employs Storj as an example technology, due to its popularity and market size. Storj is a blockchain enabled system that allows people to rent out their hard drive space to other users around the world by employing a cryptocurrency token that is used to pay for the services rendered. It uses blockchain features like a transaction ledger, public/private key encryption, and cryptographic hash functions – but this work is not centered around blockchain. Our work discusses two frameup attacks, a preliminary and an optimized attack, both of which take advantage of Storj's implementation. Results illustrate that Storj allows a potential adversary to store incriminating unencrypted files, or parts of files that are viewable on people's systems when renting out their unused hard drive space. We offer potential solutions to mitigate our discovered attacks, a developed tool to review if a person has been a victim of a frameup attack, and a mechanism for showing that the files were stored on a hard drive without the renter's knowledge. Our hope is that this work will inspire future security and forensics research directions in the exploration of distributed peer to peer storage systems that embrace blockchain and cryptocurrency tokens.

Keywords: Cryptocurrency, Distributed storage attacks, Frameup, Attacks, Security, Cloud storage.

1. Introduction

Users globally have adopted cloud storage such as Google Drive, Apple's iCloud Drive, and DropBox, maintained and centralized by companies that have created business models around storing and backing up data. However, with the rise of blockchain technology, the idea of distributed systems has become a reality, challenging the notion of companies storing and having control over people's data.

Blockchain opened the door for the ability to maintain integrity and consensus of data, which has spurred innovations in cryptocurrency development, health data records, money lending, social media and other domains. The rise of platforms such as Etherium, Bitcoin and Litecoin, has stimulated ideas, and made possible contributions such as secure distributed file storage, with the major player in that area, being Storj. Known as the Airbnb of data storage, Storj is a platform that allows individuals to store their data on rented hard drive space from people's computers around the world in a secure and distributed manner through a contract-based, blockchain, with an added Storj cryptocurrency token implementation. People can pay for rented storage space using the Storj token, and individuals renting their hard drive space, can in return receive payment with the Storj token. With billions of dollars being invested in cryptocurrency, Storj has been able to become an important player, with its Storj coin reaching a market cap of $347,106,000[1] in January of 2018.

While, distributed systems enabled by blockchain technology are vastly expanding, implementations are bound to have some weaknesses, and thus, an important scientific inquiry into these systems is critical to the privacy and security of users. Do these systems open up new attack vectors to users? If so, what are they? More specifically, can users store data on computers around the world in an unencrypted manner, potentially framing users that are renting their hard drive space? Can someone who is rent-

*Corresponding author.
Email addresses: Xiaolu.Zhang@utsa.edu (Xiaolu Zhang), jgran1@unh.newhaven.edu (Justin Grannis), IBaggili@newhaven.edu (Ibrahim Baggili), Nicole.Beebe@utsa.edu (Nicole Lang Beebe)

[1] https://coinmarketcap.com/currencies/storj/

ing drive space be shown to be innocent if an incriminating attack is possible?

This brings up the old studied claim of "a trojan made me do it" by Carney and Rogers (2004), where they explored if malware could have potentially created or downloaded illicit material onto a computer, thus incriminating a user, and if this could be detected.

Our goal was to examine the efficacy of framing people using a distributed Storj system, and by creating an approach to prove that an individual was indeed framed. We did this by taking a deep under the hood examination of the Storj platform. Thus, our work makes the following contributions:

- We provide a primary account for the security investigation of this technology.
- We inspire future inquiry into other peer-to-peer cloud storage networks.
- We make forensic practitioners aware of this potentially incriminating attack.
- We offer a deep exploration of the Storj implementation.
- We show by theory and example two incriminatory attacks that allow for the insecure and clear-text storage of data on people's computers connected to the Storj network. We also discuss the variability of the constructed and tested attacks.
- We offer suggestions on how these attacks may be mitigated.
- We developed a tool for auditing and authenticating the innocence of allegedly framed victims.

This paper is organized as follows. We first start by explaining the Storj network in Sec. 2, followed by the theory for our proposed frameup attack in Sec. 3. We then present our methodology in Sec. 4 and Sec. 5, followed by our attack implementation and results in Sec. 6. We recommend attack countermeasures in Sec. 8, and conclude our work in Sec. 9. We end the paper with related work in Sec. 10 as the core of the paper is focused on the novelty of our constructed attack, its implementation and testing.

2. Storj network

Storj is an open-source peer-to-peer (P2P) decentralized cloud storage network that embraces architectural design elements from both centralized and decentralized networks. From the storage perspective, the network is considered decentralized, as the file content is segmented and distributed across multiple peers. However, Storj uses a centralized server for communication control. The centralized server handles user authentication and negotiates and facilitates encrypted file segment storage on peer storage nodes. The Storj network is comprised of several different units shown in Fig. 1, which are the *Bridge*, *Renter* and *Farmer*.

A renter is a user that wants to 'rent' storage space on the Storj network. Renters use Storj's *Client* application to interact with the network; allowing files to be uploaded and downloaded to and from the cloud. Whenever a renter wants to communicate with the network they must first interact with the bridge. After that conversation, the bridge grants the renter approval to transfer files to and from farmers.

The bridge is the heart of the network. Every element in the network interacts with the bridge and all forms of communication are delegated by the bridge, with the exception of files transfered between renters and farmers. All renters and farmers gain access to the network through the bridge. Periodic status checks on the network are also performed by the bridge by observing all the connected farmers and renters.

Farmers are users on the network that offer space for cloud storage. Before they can provide space they must ask the bridge for permission to join the network. Once farmers have joined, they can start establishing storage *Contracts* between themselves and renters, allowing farmers to offer drive space to renters.

With the major aspects of the network explained, the communication process between a renter and the rest of the network can be described. We will now describe the procedures of uploading and downloading files to and from the network.

There are several steps involved in the process of a renter uploading a file to farmers on the P2P storage cloud. A renter must first establish a storage contract(s) with farmer(s) before the file can be handled. After the necessary contracts are in place, they are stored on the bridge and the file is queued for uploading. This queuing starts with the renter encrypting the file and then segmenting the file into different pieces called *shards*. After the shards are created, they are distributed to the farmer(s) with contract(s). Then, redundant shard copies are created and distributed as a backup mechanism, to ensure data availability, in case shards are lost or destroyed by farmer(s), or the farmer(s) are off-line when the renter wishes to access their file.

When downloading a file, the renter contacts the bridge to request the file from a farmer(s). The bridge first checks if the renter can download the file by verifying if the file can be reconstructed from available shards. If the file can be reconstituted, then the bridge tells the farmer(s) to start sending the shards back to the renter. Once the renter has all of the shards necessary for restoring the file, the shards are combined into one file and decrypted. At this point, the file has been retrieved and is stored on the renter's computer with the bridge auditing the transaction.

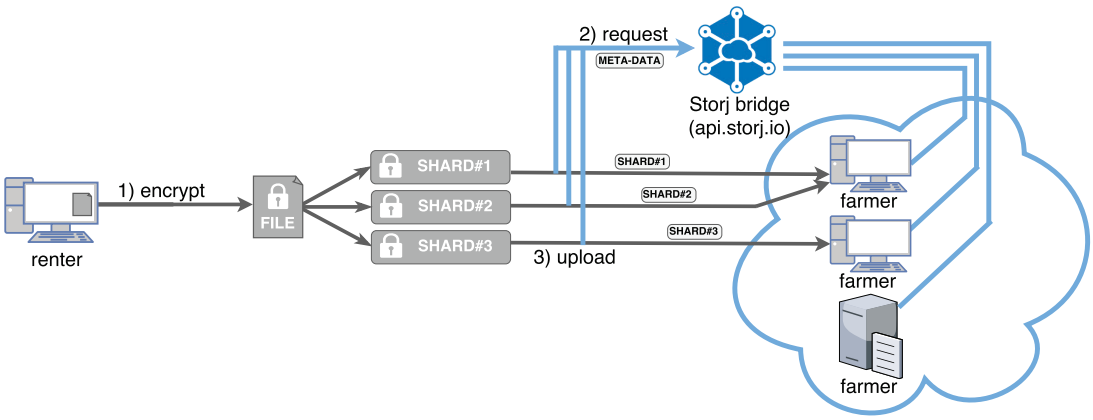

Figure 1: Upload procedure in Storj network

3. Frameup attack

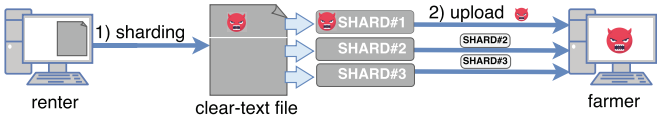

Figure 2: Process of frameup attack

In this section, we identify and explain an important security issue posed by decentralized storage. This issue has been overlooked to date, as extant research has focused on centralized storage networks. Decentralized storage involving individual users' personal computers provides new vectors adversaries may exploit. We term this attack the *Frameup* attack and depict it in Fig. 2.

In this attack, nefarious Storj renters can upload unencrypted files to farmers' computers around the world, potentially consisting of malicious software, contraband material, and other content with malicious intent. This is made possible by a renter disabling the encryption process prior to file segmentation and shard uploading. This can result in unencrypted data being stored on farmer computers, which could then be unwittingly executed in the case of malware, or unknowingly possessed in the case of contraband. It is important to caveat, however, that a nefarious renter is not able to target specific farmers with the frameup attack.

While not being able to distinctly target individuals may limit the scope of the attack to non-targeted victims, target victimization is achievable. Based upon Storj's design, their protocol for selecting a node/farmer for storing a renter's shard is deterministic. A farmer is nominated to store a renter's shard when they have the lowest latency and fastest data transfer rate to the renter. An informed attacker can leverage this design characteristic to increase the precision of their attack. If the attacker and target are geographically within the same network or facility, then a widely distributed attack on the entire cloud is not necessary.

Performing a frameup attack may not always be limited to a single entity/person/computer. A wide range of malicious activities produce a great amount of harm. Renters may specify the amount of redundant copies of their file to be stored on a Storj network. Therefore, investigators that discover a malicious shard on a farmer may identify that the shard was mirrored to hundreds or thousands of other farmers. This situation is achievable and problematic because it may induce forensic examiners to consume an immense amount of time acquiring and examining all affected farmers.

4. Methodological overview

To validate the incriminatory frameup attack we needed: 1) a live Storj storage network where all the components such as farmers, bridge, complex, and renter nodes can be experimentally monitored; 2) Storj client software with which we could disable file encryption and upload files to the Storj network; 3) mock incriminatory files of various sizes and types (e.g., text files, images, videos, executables etc.); and 4) digital forensic software to conduct a forensic analysis of hard drives involved. We successfully constructed the attack using the following steps:

1. Because not all network components could be monitored and analyzed if we used a Storj maintained bridge, we built our own local, private Storj network. However, because we used Storj open-source code[2], we believe the experimental findings on our network are generalizable to public Storj networks. See Sec. 5 for details.

2. To facilitate farmer storage of clear-text shards, we first configured the Storj Client to connect to the private network and then modified the uploading behavior of the client source code. The modification is described in Sec. 5.2.

[2] https://github.com/storj

3. Further explained in Sec. 5.4, we then created a clear-text file data set which included different types of documents and multimedia files. We then, as described in Sec. 6.1, uploaded the clear-text files in a range of sizes to verify if the farmer stored the same content as the original file being sharded.

4. Based on results obtained in step 3, the frameup attack was optimized. The optimized attack encapsulated the uploaded clear-text files in HTML to better survive the file sharding process. Further elaboration is provided in Sec. 6.2.

5. We then verified the attack from the forensic investigator's perspective using the widely adopted forensic tool Forensic Toolkit (FTK)[3]. FTK's data carving process enabled us to verify the clear-text shards were recoverable, and in the case of executable content, could be encapsulated in such a way as to execute on the forensic station. The specifics are presented in Sec. 7.

6. Due to the frameup attack, we presented an approach that can test if the uploaded shards contain unencrypted data. Additionally, we illustrate a strategy for those who have been framed. The strategy can show that the victim was under a frameup attack from a technical perspective. These prevention and investigation approaches are shared in Sec. 8.

5. Detailed methodology

5.1. Private Storj network

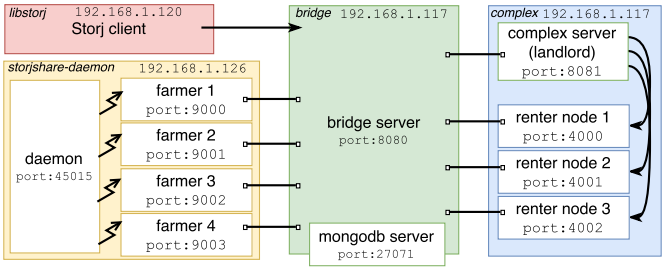

Figure 3: Components in private Storj network

The experimental environment required a private Storj network for cloud storage and a nefarious renter's client application requesting the storage service. As shown in Fig. 3, the red block represents the nefarious Storj renter and client software. We labeled the software project names at the top left corner of the blocks. For example, in order to build a bridge server we downloaded the project of 'bridge' from Storj Labs' Github page and customized it. Note that both the components of the private network and the client application connecting to it were hosted on Ubuntu 16.04 LTS computers in the same local network. The *bridge* and *complex* were deployed on the same computer. The client and farmers were installed on other local network computers respectively.

As Fig. 1 depicts, the Storj network needs a centralized server (*bridge*), to make contracts for renters and farmers and store essential information such as registration for renters, node IP address, contract details, meta-data of shards, etc. In our private network, a bridge (v5.22.2) server was assigned IP:port `192.168.1.117:8080` instead of `api.storj.io`, which is the bridge server for the public Storj network. Accordingly, the value of string 'host' was replaced with `http://192.168.1.117` in the object 'server' in the configuration file of the bridge server, which is the Json file `~/.storj-bridge/config/develop` by default. We also hosted a MongoDB server on port 27017 to support the infrastructure for the private bridge.

From a programming point of view, Storj labs is considered a Complex server and a few renter nodes are necessary components for the Storj network. The Complex is similar to the landlord of an apartment complex. They manage *renter nodes* on the Storj network. The renter nodes are not the same as renters; they do not request storage space. Their only purpose is to provide a gateway for farmers and renters to join the network at anytime. That is to say, they are always joined to the network to keep the network alive. In essence, a Complex contains 'always known' access points for joining the network. Theoretically, if the bridge has known enough farmers before initializing the network, the Complex server or the renter nodes would not be needed. However, in reality, the bridge always needs a few nodes to generate the original Storj network for other nodes / farmers to join. What's more, the renter nodes can be created by Storj labs or others who want to initialize a Storj network. The Complex server's purpose is for managing them. In our local network, these components were constructed by configuring the Complex (v5.6.0) project. We assigned port 8081 to the Complex server and port 4000 – 4002 to the renter nodes.

To provide farmers with the ability to join the original Storj network, project 'storjshare-daemon' or 'storjshare-gui' was required to be installed. In this case, we choose storjshare-daemon (v3.5.5), which has a Command Line Interface (CLI). As the project was installed on computer `192.168.1.126`, a valid Ethereum (ETH) wallet address was fed to the CLI command 'storjshare-create' and the configuration file for each of the four farmers were generated in the folder `~/.config/storjshare/configs/`. The configuration file for the farmer was named by its node ID, such as `68336ce3f8b3ad5052c4259bbbde707057ee8cb2.json`.

Afterwards, we modified the configuration files. In particular, 'bridgeUri' was assigned to `http://192.168.1.117:8080`, enabling the farmers to connect to our private network. Also, 'seedList'

[3]FTK is a commercial forensic tool that was wildly used for forensic investigators and law enforcement agencies.

was populated with the information of the renter nodes. For instance, `storj://192.168.1.117:4000/337472da3068fa05d415262baf4df5bada8aefdc` is a valid representation for a renter node at port 4000. Note that a filled 'seedList' would help a farmer become a node of the initialized private network. Additionally, a 'daemon' program that provides remote control on farmers, usually runs before initializing the farmers. Once the farmers became nodes, the private Storj network was established.

In order to provide the uploading and downloading functionality for renters, Storj Labs released the project 'libstorj' as a client application that was integrated based on the Storj API. Given that we introduced the modification of 'libstorj' in Sec. 5.2, at this point of time, it should be noted that the unmodified version of 'libstorj' was cloned to the local network computer at `192.168.1.120`.

5.2. Modification on Storj client

To allow the Storj client, 'libstorj', to upload a clear-text file to our private Storj network, the client must be able to find our private bridge. Thus, as shown in Listing 1, we first modified line 1155 of the main function in file `~/libstorj/src/cli.c`, which forwards the client to our private bridge at `192.168.1.117:8080`

```
if (!storj_bridge) {
    storj_bridge = "http://192.168.1.117:8080/";
}
```

Listing 1: Setting 192.168.1.117 as bridge

Second, we found three functions related to the encryption of the uploaded file where function `prepare_frame(..)` and `create_encrypted_file(..)` were stored respectively on lines 1428 and 1603 in the file `~/libstorj/src/upload.c`; the function `body_shard_send(..)` was stored on line 37 in the file `~/libstorj/src/http.c`. All of these functions called another function `ctr_crypt` to encrypt files. Since the function `ctr_crypt` was implemented in all three encryption functions in a similar manner, we only illustrated Listing 2, belonging to the function `prepare_frame(..)` as an example.

We modified all three of the functions in this case. Once calling `ctr_crypt`, the AES256 encryption was performed on the clear-text data stored in variable `read_data`, as well as the encrypted data stored in variable `cphr_txt`. Since we intended to upload the clear-text data rather than the encrypted data, the function `memcpy` was inserted following the encryption function in order to replace the encrypted data back to the clear-text. In other words, although the Storj client successfully conducted the encryption function, the data finally uploaded to the Storj network was unencrypted.

```
// Encrypt data
    ctr_crypt(encryption_ctx->ctx, (
    nettle_cipher_func *)aes256_encrypt,
    AES_BLOCK_SIZE, encryption_ctx->
    encryption_ctr, read_bytes,(uint8_t *)
    cphr_txt, (uint8_t *)read_data);
// Replace the cypher-text to clear-text
    memcpy(cphr_txt, read_data, AES_BLOCK_SIZE
    *256);
```

Listing 2: Replacing cipher-text with clear-text

Once the client was modified, it was compiled for uploading an unencrypted version of a file to the private network.

5.3. Account registration

To upload a file to the Storj network, a user must have registered an account and password on the bridge. Along with a valid user account, the system must have at least one virtual bucket for accommodating the uploaded files. The unique ID of the bucket must be provided for file uploading. For example, in our case, we created the bucket `a01501b963e7b23e9203d206`. Command `storj upload-file a01501b963e7b23e9203d206 <file_name>` can be utilized for uploading files to the virtual bucket. Along with that, the Storj client provided the command `storj list-mirrors <file_id>` for tracing which farmer stored the shards.

5.4. Storj files and data structures

To test the frameup attack, a number of clear-text files were needed. However, in order to construct a dataset of the clear-text files, Storj data structures must be understood.

Storj implements Google's LevelDB for storing shards on farmers within a file storage system called Kademlia File Store (KFS). LevelDB contains .log and .ldb database files for storing data, which is where Storj stores the content of shards on the farmers. The .log file contains the most recent updates/additions to the database. When the .log file reaches a pre-determined size of 4 MB the file is converted over to a .ldb sorted table, which also has a pre-determined size of 4 MB.

Within the .log and .ldb files, there are 'extra' bytes inserted into the file that are not from the original sharded file. These extra bytes are intended to be within .log and .ldb files because they are related to the LevelDB file structure. There are two forms of inserted bytes that appear in a .log file and two forms in a .ldb. For .log files: a group of 69-71 bytes appear about every 128KB and a group of 7 bytes every 32 KB. For .ldb files: a group of 58-60 bytes appear at the beginning of the file and a group of 71-73 bytes appear about every 128KB. The 7 byte sequences are derived from the specifications of Leveldb, as follows. Every 32 KB of a .log file is considered a 'block'. Each 'block' consists of a sequence of 'records': the crc32c checksum in little endian of the 'data', the length of the data in little endian, the 'block' type, and a sequence of bytes called 'data'. The 7 bytes are derived from the 4 byte crc32c, 2 byte 'data' length, and the 1 byte 'block' type. Following the 7 bytes is a sequence of 32 KB or less from a shard. If there is a record with a 'data' section less than 32KB in a 'block', then it is possible for another record to reside

within said 'block'. Therefore, it is possible for one .ldb or .log file to contain 'blocks' from multiple shards/files.

Every 128 KB of a .ldb and .log file is considered a 'chunk' of data from a shard. A 'chunk' can be less than 128 KB, which means that .ldb and .log files can be less than the size of a chunk. Note that a 'chunk' may start slightly after the 128KB mark to offset the addition of n instances of inserted byte groupings from earlier 'chunks'. These 'chunks' contain a unique identifier for the purpose of efficiently finding every chunk of a shard when a download request, from a renter, has been made. At the beginning of a chunk in a .log file, there is a key, which is a group of 69 to 71 bytes of data containing: the 7-byte grouping, a byte for indicating the chunk's number/identifier in the .log file, the 12-byte sequence '0x000000000000000100000001', a forward slash followed by the full content's hash formatted in hexadecimal, a space, a 6-byte numerical index, and 1 to 3 bytes that relate to the chunk's length. At the beginning of a chunk in a .ldb file, there is a 9-byte sequence '0x000000000100000000', followed by a 4-byte sequence, followed by '0x0037' followed by a 1-3 byte sequence related to the length of the chunk, a 40-byte key for the file that the chunk relates, a space, a 6-byte chunk index identifier, a 2-byte value that indicates the chunk index identifier in hex format, followed by the 6-byte sequence '0x000000000000'.

With this foreknowledge of KFS in-hand, the clear-text files used for our test were selected from the *GovDocs*[4] dataset where the entire file could fit within one shard. A shard has to be 4197472 Bytes (about 4.1 MB), including the extra bytes. We collected 20 files of each of 15 widely used file types including: JPG, GIF, PNG for pictures; AVI, FLV, MOV, MP4, MPG for videos; TXT, DOC, PDF, XLS for documents and ZIP, GZIP, BZ2 for compressed files. 300 total files were collected and can be found at the GitHub repository[5].

6. Frameup attack testing and results

This section presents the complete frameup attack process in the Storj network. First, we present our preliminary attack, used to verify the ability to upload non-encrypted files to farmers. Next, we explain how we continuously optimized the attack to increase its range and impact. For both attacks, we uploaded the clear-text files from the test file dataset and manually verified whether the shards were recoverable, visible, and/or executable on farmer nodes.

6.1. Preliminary attack

The preliminary attack is the initial phase of the frameup attack. In this attack, we intended to determine: 1) if the modified version of the Storj client, designed to revert back to the clear-text version of the original file upon upload; 2) if the content of the clear-text files could be found in shards; and 3) if the clear-text files could be opened and viewed on the farmer.

Figure 4: Original picture Vs. the 'recognizable' shard

First, we registered and made available two farmer nodes. We then uploaded clear-text files from the test dataset to our private Storj network by scripting the command `storj upload-file a01501b963e7b23e9203d206 <file_name>` (`a01501b963e7b23e9203d206` is the bucket ID) for each file. After successful upload notification, we accessed the folders `~/.config/storjshare/shares/68336ce3f8b3ad5052c4259bbbde707057ee8cb2/sharddata.kfs` and `~/.config/storjshare/shares/d1f5b48e687be49f9472c3eeca0099030cc8ae6b/sharddata.kfs` on computer `192.168.1.126` of our private Storj network for examining the shards, in which `68336ce3f8b3ad5052c4259bbbde707057ee8cb2` and `d1f5b48e687be49f9472c3eeca0099030cc8ae6b` refer to the node IDs. Within the folders there were hundreds of sub-folders named as 'xxx.s' where 'xxx' was a unique number generated by the farmer. Each of these numeric folders stored the shard file(s). To verify the preliminary attack we manually examined the shard files stored by both of the farmers.

As expected, we found the content of the uploaded files in clear-text in the shards along with the extra KFS generated data described previously. Next, we manually tested how many of the shards qualified for the frameup attack by being recoverable and viewable. To reduce the workload, we identified and leveraged file signatures for most of the shards. For example, if the first bytes in a shard were `JFIF`, the shard would be considered a `.jpg` file, and an appropriate graphic file API was used to open the file. However, as we explained previously, a shard may contain data from multiple files. Therefore, where shard files did not begin with a known file signature, we tried different file extensions before dragging them to an executor[6]. Note that we only tested if a shard could be executed or materially recognized, such as the partially corrupted picture (shard) shown in Fig. 4, which was still 'recognizable' by investigators.

[4] https://digitalcorpora.org/corpora/files
[5] https://github.com/jgran1/Storj_Test_Data/tree/master/Dataset
[6] In the preliminary attack, we achieved the test on a Windows 7 Enterprise SP1 PC, where Windows Photo Viewer, Windows Media Player, Windows Notepad v6.1, Microsoft Word 2016, Microsoft Excel 2016, Adobe Acrobat Reader v17.012 and 7-Zip v16.04 were utilized as the executors.

Table 1: Summary of readable / recognizable shards for the Preliminary Attack

Type	Farmer 68..	Farmer d1..
JPG	0/20	0/20
GIF	0/20	0/20
PNG	0/20	0/20
AVI	14/20	13/20
FLV	0/20	0/20
MOV	3/20	3/20
MP4	0/20	0/20
MPG	20/20	20/20
DOC	0/20	0/20
XLS	0/20	0/20
PDF	10/20	10/20
ZIP	0/20	0/20
GZIP	0/20	0/20
BZ2	0/20	0/20

The majority of the shards were not recognizable. The extra data added during the sharding process rendered the files unrecognizable. For instance, the JPG file structure contained segments with markers identifying its meaning. If the marker is modified (or shifted) then the meaning of the segment following the marker may cause the file to become corrupted. Only a few file types such as AVI, MPG and PDF produced relatively high executable rates. The number of the executable shards for different types of files are listed in Table 1.

Note that even though we found the content of TXT files in the shards, it is possible for the data of one TXT file to be separated between multiple shard files. Hence, while we conclude TXT files are eligible for the preliminary attack, we did not count them in Table 1.

While our experimental findings verify that attackers are able to upload clear-text files to farmers unwittingly, as Table 1 shows, the attack is only effective from a readability standpoint for a few file types. That is, many file types cannot withstand the extra bytes added during the sharding process. Accordingly, we propose an optimized attack in Sec. 6.2 that can better maintain file content integrity and usability.

6.2. Optimized attack

Theoretically, even though the files with extra data inserted become corrupted, the data is not lost. Rather, the data is being shifted around. Therefore, if the file is treated such that there is a logical split at these insertion points then it is possible to combine these parts into the original file while ignoring the extra data causing the problem. To bypass the embedded data in the opening and execution of the shards, we propose HTML encapsulation.

Specifically, we embed the test file into a HTML file by converting and encoding the desired file into base64 plaintext. We pre-calculate where the extra data will be inserted and prefabricate a comment statement at the appropriate place to convert the extra data into a valid HTML comment that will be ignored upon opening. Therefore, the extra data will no longer corrupt the file's readability. Further, we store the file content in string variables and use Javascript to open the file content. This is shown in Listing 3. Here, Javascript provides the means to segment the base64 encoded file into separate string variables. Separation between two string variables occur near the locations where extra bytes are inserted during the sharding process. The text area between two string segments is identified as '`Byte Injection Space (BIS)`'. BIS is a section of text where we intend to handle the extra data. Therefore, in this case, it must occur every 0x8000 bytes (32 KB).

In order to include a BIS without affecting the base64 encapsulated data we surrounded a BIS section with the comment keywords '`/*`' and '`*/`'. By commenting the BIS almost all injected byte combinations (except for the extremely rare premature occurrence of '`*/`' that will end the BIS early) will be interpreted as meaningless text. Once the comment section is placed around the BIS, a large padding section of text, such as a long sequence of '#', is inserted to provide a large safety net for containing the inserted bytes. In addition to the variables `str0` and `str1`, there are HTML tags and attributes which will be recognized by the browser once the Javascript code is interpreted. Tag '`<img`' defines an image in an HTML page. Its attribute '`src`' must contain the text '``', where '`image/format`' is the image file type. This is so the base64 encoded data can be decoded into the original file format and be displayed on the HTML page.

```
<html>
<script language="Javascript">
var str0 = "<img src=\"
 iVBORw0KGgoAAAANSUhEUgAAAM0AAADNCAMAAAAsYgRbAA
 AAGXRFWHRTb2Z0d2FyZQBBZG9iZSBJbWFnZVJlYWR5ccll
 PAAAABJQTFRF3NSmzMewPxIG//ncJEJsldTou1jHgAAAAR
 BJREFUeNrs2EEKgCAQBVDLuv+V20wKwKz831wKz4/VHb
 /5RGQONDQONDQONDQONDQONDQONDQONDQONDQONDQO
 NDQOPzMWtyaGhoaGhoaGhoaGhoxtb0QGhoaGhoaGho
 aGhoaGhoaMbRLEvv50VTQ9OTQ5OpyZ01GpM2g0bfmDQaL7
 S+ofFC6xv3ZpxJiywakzbvd9r3RWPS9I2+MWk0+k"
/*
#### Byte Injection Space ####
##{[8|ï£¡-/53853dc1313238a0a508bba3b9f274e3a3fb3
a4f 000000ï£¡ï£¡############
*/
var str1 = "bf0Hih9Y17U0nTHibrDDQ0NDQ0NDQ0NDQ0
 NDQ0NTXbRSL/AK72o6GhoaGhoRlL8951vwsNDQ0NDQ1ND
 c0WyHtDTEhDQ0NDQ0NTS5MdGhoaGhoaGhoaGhoaGh
 oaGhoaGposzSHAAErMwwQ2HwRQAAAAAElFTkSuQmCC\">"
document.firstline = str0 + str1 + "\n"
</script>
<body>
<script>
document.open()
document.write(document.firstline)
document.open()
</script>
</body>
</html>
```

Listing 3: Base64 encoded image in HTML with string segmentation

The base64 plaintext data encapsulated in HTML will

not be impacted by the added data and the image is displayable on Internet Browsers. However, due to the fact that not all types of files are displayable on browsers, we added a feature to handle file types that are not suitable for display in such a browser. Here, the uploaded shard consists of an HTML file designed to open a file in a browser that either automatically downloads the target file from a web server, or prompts the user with a local download request. We applied this file restoration technique for the following file types: AVI, FLV, MOV, MPG, DOC, XLS, PDF, ZIP, GZIP and BZ2. This technique could also be used to upload files larger than the 4MB shard size limit imposed thus far.

```
<!DOCTYPE html>
<html>
<body>
<script>
function base64Img2Blob(code){
// convert Base64 data to Blob data for IE.
  var parts = code.split(';base64,');
  var contentType = parts[0].split(':')[1];
  var raw = window.atob(parts[1]);
  var rawLength = raw.length;
  var uInt8Array = new Uint8Array(rawLength);
  for (var i = 0; i < rawLength; ++i) {
    uInt8Array[i] = raw.charCodeAt(i);
  }
  return new Blob([uInt8Array],{type: contentType
      });
}
function detectBrowser() {
//detect which Browser the HTML is executed on.
  var userAgent = navigator.userAgent;
  if (userAgent.indexOf("Firefox") > -1) { return
       "Firefox"; }
  if (userAgent.indexOf("Trident") > -1) { return
       "IE"; }
  if (userAgent.indexOf("Chrome") > -1) { return
      "Chrome"; }
}
var blobObject =  new Blob( [base64Img2Blob("
    ...")]);
// base64 data of the uploaded file.
if (detectBrowser() === "IE") { //If executed on
    IE
  window.navigator.msSaveOrOpenBlob(blobObject, '
      example.png');
} else { // If executed on Firefox or Chrome
  url = window.URL.createObjectURL(blobObject);
  a = document.createElement('a');
  a.download = 'example.png';
  a.href = url;
  document.body.appendChild(a);
  a.click();
}
</script>
</body>
</html>
```

Listing 4: Javascript code for files embedded into HTML file to be automatically downloaded on browser

Listing 4 shows an example of the HTML file with the downloading feature. Since Internet Explore (IE) does not support anchor (<a>) tagged elements[7], our code was configured to detect if the browser is IE by the function `detectBrowser()`. For the IE browser, we needed to supply the base64 data to the function `base642Blob()` for converting the data to a Blob object followed by passing the Blob to the function `window.navigator.msSaveOrOpenBlob(..)` to utilize the download feature. For other browsers we created the <a> element for downloading.

To test the optimized attack, we activated the other two farmer nodes created during the preparation phase. The ID / folder name of the farmer nodes are `5a2f43355526137739603667241 8f765b51f0de9` and `f33e0bfe6eaa263bf2eb08b9d919c80ac5c30157`. Next, we converted the clear-text files to their proper HTML equivalents and uploaded them to the private Storj network.

To test whether the embedded file can be displayed or downloaded on the browser in this manner, we collected all the shards on the two farmer nodes and changed the extension of the shards to '.html' and attempted to open them with IE. Our results are shown in Table 2. We see that the optimized attack is viable for uploading and successfully reading all file types, although not in all cases. On average, more than 55% of the uploaded files can be located and recovered by executing the shard files. Note that, different from the preliminary attack, the shards that are successfully downloaded and displayed in the browser, not all maintain their full, original integrity.

Table 2: Result summary for the optimized attack

Type	Farmer 5a..	Farmer f3..
JPG	12/20	13/20
GIF	12/20	11/20
PNG	13/20	12/20
AVI	8/20	8/20
FLV	11/20	9/20
MOV	10/20	10/20
MP4	11/20	11/20
MPG	3/20	3/20
DOC	12/20	13/20
XLS	12/20	12/20
PDF	12/20	12/20
ZIP	12/20	12/20
GZIP	13/20	14/20
BZ2	14/20	15/20
Total	55.36%	55.36%

7. Attack evaluation with FTK

In Sec. 6, we tested the recognizability of the file shards using both the preliminary attack and the optimized at-

[7]The HTML <a> element A.K.A "archer element" creates a hyperlink of the files that intended to download on the HTML page. Using the `download` attribute user will receive a promote from the browser for download. Element <a> is not supported by IE but Chrome and Firefox.

tack. However, manually testing all the evidence is not standard practice for real-world forensic investigations. Therefore, in this section, we tested and evaluated the frameup attack by loading the shards in FTK 6.2.1. Note that even though FTK provides a variety of features, our test mostly relies on the data carving feature pertaining to the recovery of clear-text files or mostly visible partial clear-text files. Ultimately, we empirically determined: 1) how many of the clear-text files can be recovered by FTK, and 2) the difficulty of the process related to the discovery of clear-text files, within a shard, in real-world investigations, using FTK.

In FTK, once the 'data carve' function is activated in 'Processing profile', ideally, it will carve out the supported type of files from the shards. Out of our dataset, the supported types of files are JPEG, HTML, PDF, GIF, PNG and ZIP. Since FTK has a built-in IE browser API, an investigator can preview the carved clear-text files on the screen. When compared with the manual test (result shown in Table. 1), FTK retrieved more clear-text files from the shards. To be specific, the same 12 DOC, 17 JPG, 11 PDF, 4 GIF, 15 PNG and 10 XLS executable files were carved out from both of the farmers. Note that the files were partially corrupted, but in order to be counted as carved we used the threshold of at least 50 percent or more visible content to the end user.

On the other hand, when processing the shards created by the optimized attack, FTK was not able to retrieve any clear-text files. When we used the built-in IE API in FTK to display the file content, Javascript did not run. Thus, in order to execute the shards for extracting the clear-text files, the investigator would have to either export the carved file shards and execute them with a browser external to FTK, or the investigator would have to double-click on the file in the file listing view, which by default, causes FTK to execute the file via an external browser, thus causing the Javascript to execute. With this approach, we gained the same result as the manual test (see Table. 2).

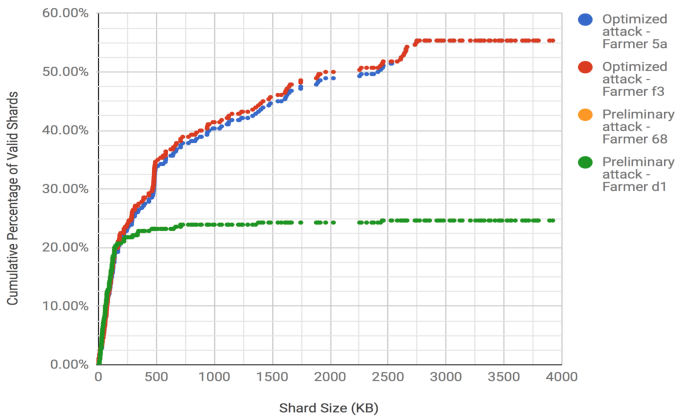

Figure 5: Preliminary attack vs. optimized attack

In Fig. 5, we demonstrate the results of the FTK test. Here, the X-axis shows the size range of the clear-text files (shard files) in our dataset (TXT files are not included) and the Y-axis is the cumulative percentage of files, up to that shard size, that are valid for these two attacks. The green dots represent a farmer under the preliminary attack. For files of size 500 KB or less, 23.21% of all files were valid. For files of size 1000 KB or less, 23.93% of all files were valid. The number of the valid shards becomes asymptotic when the clear-text files reach to 500 KB. With that said, the optimized attack did a better job for the larger clear-text files and it also attained a higher success rate in general.

In summary, the FTK tests disclosed the following attack advantages and disadvantages:

Preliminary attack:

- Advantage: no extra step for investigators to discover the clear-text file, as long as a forensic tool like FTK is used.
- Disadvantage: 1) only works for fairly small sizes of clear-text files, 2) relies on the data carving feature of the forensic tool, which may not natively support all file types of interest, and 3) even though some shards are valid for the attack their integrity can not be guaranteed.

Optimized attack:

- Advantage: 1) works equally well for varying sizes of clear-text files, and 2) executable code (e.g. malicious Javascript) can be uploaded to farmers with ensured integrity.
- Disadvantage: 1) HTML file, as well as the Javascript, need to be executed in a browser external to the forensic tool to ensure all file effects are realized.

8. Countermeasures

In this section, we focus on the prevention of the frameup attack where we present two approaches for either detecting if a farmer node contains any uploaded clear-text files or proving that the improper files examined from a framed computer / server were not stored under the awareness of the farmer.

8.1. Attack detection

Storj and other decentralized cloud storage providers might implement a variety of attack detection techniques to mitigate the attack risk discussed herein. First, they might modify farmer client software to include an entropy test on data to be stored and refuse the storage of data that fails the entropy test. However, given the high entropy associated with executable, zip, and various graphic file types, the entropy threshold should be set very high. This technique might be improved by a sliding-window based approach to entropy calculation, according to Hall

et al. (2006) and Beebe et al. (2013). In the case of properly encrypted shards, the entropy of the full shard file should be relatively equivalent to the entropy of blocks within the shard. A shard file consisting of blocks with varied entropy levels may indicate that it is not properly or fully encrypted, or that it is of a different high entropy file type, such as those listed above.

Another solution, independent of, or perhaps in conjunction with the entropy test, would be a signature based approach to misuse detection. Each shard could be scanned for common HTML tags or other strings common to popular applications, including but not limited to Internet browsers. Perhaps farmers could be trained through Bayesian or other probabilistically based supervised learning techniques, to detect nefarious content via n-grams and block-level entropy measures as suggested by Hall et al. (2006). When such signatures are found and/or content is classified as nefarious, the farmer can and should reject the uploaded content and log the action and rationale thereof on the centralized server.

A third solution would be for Storj to modify its renter application to *not* permit a user to disable encryption of the file prior to the segmentation and uploading of shards. However, since the code is open-source, a technically adept adversary could simply re-instate the user ability to disable encryption.

Note that adding the aforementioned detection approaches on the farmer side would provide minimal effort to prevent the attack. However, that way, the detection would only take place after the contract is made, which means a legal contract would be rejected by farmers if the attack is detected. Also, before the rejection happens, all benign shards of the uploaded file will have been transferred to the farmer, which consumes network traffic. Therefore, ideally, the entropy / the signature should be calculated and put in the contract for both sides to verify. However, it may cause an increase of heavier workloads on the decentralized system. In its current implementation, it is difficult to mitigate our presented incriminatory attack without finding a balance between efficiency and complexity.

8.2. Forensic analysis on farmer defense

As previously discussed, clients have the ability to upload unencrypted shards to the network, which may contain unwanted information for the purpose of framing farmers. Therefore, devising a method of protecting a farmer is paramount. This section will layout a method of gathering evidence for defending a farmer, which requires access to information on both the bridge and farmer. The high-level perspective of the process is to: 1) acquire and extract the *shard* table from the database in the bridge on the network, 2) generate the hash of the 'questionable' shard on the farmer to verify the hash stored in the bridge's database, 3) locate either a *SHARD_UPLOADED* or *MIRROR_SUCCESS* entry in the bridge's database to further verify the hashes, 4) find information about the client that uploaded the 'questionable' shard, and 5) gather information about other farmers that have a 'mirrored' (backup) 'questionable' shard. In our case, the forensic process of defending a farmer will be described through a tool[8] we have developed, based on the implementation of Algorithm 1.

Algorithm 1 Verification Algorithm

```
bridgeShardHashList ⇐ getBridgeShardHashes()
fShardHashes = {}                          ▷ Farmer shard hash list
for all FarmerLevelDBDatabases do
    for each shardData ∈ FarmerShardsInDB do
        shardHash ← computeShardHash(shardData)
        fShardHashes ← fShardHashes ∪ shardHash
    end for                                ▷ Above: Insert hash into list
end for
validHashes = {}, invalidHashes = {}
for each shardHash ∈ shardHashes do
    if shardHash ∈ bridgeShardHashList then
        validHashes ← validHashes ∪ shardHash
    else
        invalidHashes ← invalidHashes ∪ shardHash
    end if
end for
printResults(validHashes, invalidHashes)
```

Starting the defensive process requires the acquisition of all shard hashes on the Bridge's database. This can only be achieved by contacting the owner of the bridge because the database is not accessible outside of the bridge computer. The bridge's IP address was used for this purpose and its IP address can be found under the farmer computer in its configuration file `/.config/storjshare/configs/<farmerid>.json` under the *bridgeUri* configuration setting. Having the bridge hashes of shards provides the means of linking metadata of a shard on the bridge's database with shard information on the farmer. In this case, the goal is to verify the integrity of a 'questionable' shard. This can be done by connecting to the database through a MongoDB viewer.

Since we controlled the bridge, we accessed the database by using the mongodb viewer *Robo 3T 1.1.1* on the bridge computer, where the database was stored under the directory `/var/lib/mongodb`. Refer to Appendix A for sample metadata from a farmer's configuration and log files and the bridge's database. This application makes it possible to extract all the JSON information regarding to shards that have been transfered through the network. The JSON information of shards on the database contains hashes of each shard's contents. Once access to the Bridge's database is obtained, then the *shards* table from the database can be extracted to a file in JSON format. Once extracted to a file, the developed tool will read the contents of the file and load the database hashes for further

[8] https://github.com/unhcfreg/Frameup

processing. Listing 5 shows the beginning of an extracted shard record from the *shard* table in JSON format and it contains the hash value for a shard.

```
/* 1 */
{
 "_id" : ObjectId("5a09d54f2d579c7a1ea0087e"),
 "hash" :
  "40f892fc2c2a5b5fac320d6154da6b8919246db1",
 "meta" : [
  {
   "nodeID" :
    "5a2f433555261377396036672418f765b51f0de9",
   "meta" : { "downloadCount" : 0 }
  }
 ],
  ...
```

Listing 5: Snippit of the Bridge's shard table from the database

Extracting the contents of the questionable shard is required to generate the hash of the shard on the farmer's side and verify that it has not been modified or generated by the farmer. This was achieved by using *plyvel*, a python library that can interact with LevelDB database files/directories. The developed tool will traverse the LevelDB directories, which contain shards, on the farmer's share files/directories. An example directory structure may look like the following: `/shares/<farmernodeid>/sharddata.kfs/*`. For every shard found in the database, the extraction process will be executed to gather the contents of the shard.

Hashing the contents of the shard follows the extraction process and is uniquely labeled in a dictionary within the developed tool. A hash of shard is generated by taking the SHA256 checksum of its content followed by taking the RIPEMD160 checksum of the binary hexidecimal output from the SHA256 checksum. Once all of the shards have been hashed from the farmer the tool will determine if the shards have been tampered with. Verifying a shard's integrity from the farmer is conducted by checking if the calculated shard hash exists in the *shard* table of the bridge database. The tool then finishes by displaying all of the hashes that have been verified from both the farmer shards and the bridge database along with displaying unverified hashes from the farmer.

Besides for verifying the shard hashes, a bridge's database contains other valuable information that can be used to further support the defense of a farmer. There are 16 *collections* in the database, where the *exchangereports* and *mirrors* collections hold relevant information pertaining to a farmer's defense. Each collection consists of multiple *keys* where each key is made up of multiple *fields*. Along with that, each key in a collection contains the same number of fields. Knowing the database structure, the process of verifying the shard metadata and the farmer can be executed. There are three main goals to verifying a key to defend a farmer: 1) the shard hash in the key and the farmer, 2) the farmer's ID in the key and the farmer and 3) the *exchangeResultMessage* field to match either *SHARD_UPLOADED* or *MIRROR_SUCCESS*.

Other tables in the database contain shard hashes, which can be used to provide additional support when verifying calculated shard hashes on a farmer. Furthermore, this also shows that the farmer was meant to receive the shard. Verifying the hashes was performed by matching the hash of the shard from the farmer to a *dataHash* field in a key of the *exchangereports* collection of the bridge's database. Confirming the hashes shows that the farmer has not modified the shard data. Next, the farmer's ID was verified by matching either the *reporterId* or *farmerId* fields, in the same key where the hash was verified, to the ID contained in the farmer's configuration file. Lastly, the *exchangeResultMessage* field, in the same key, was verified to be *UPLOAD_SUCCESS*, which means that the farmer has received that shard successfully. By verifying the *exchangeResultMessage* and either one of the *reporterId* or *farmerId* fields proves that the shard was uploaded to them from a client. Note that if the *exchangeResultMessage* field held the value *MIRROR_SUCCESS* then the shard came from another farmer as a backup 'mirror'. After verifying all three of the goals the farmer has been defended since the shard was uploaded to the farmer and the farmer has not modified the data.

Gathering as much information as possible about the shard owner is crucial for the purpose of preventing further distribution of 'questionable' shards on the network. Information about the client that uploaded the 'questionable' shard was discovered by finding the key in the *exchangereports* collection that 1) has the same *dataHash* field value as the key that verified the 'questionable' shard and 2) has the value of *SHARD_UPLOADED* in the *exchangeResultMessage* field (Note that the key may be the same). The key that satisfies this criteria contains the client's ID in the *clientID* field and the *created* field holds the timestamp of when the shard was uploaded to the network. After identifying the client's ID we viewed the *users* collection in the bridge's database and found the key that matches the identified client's ID. The user's key contains multiple fields that provide information about: the ID of a user, the user's hashed password, bytes downloaded in the last month, day, and hour, bytes uploaded in the last month, day, and hour, the time the user was created, and the uuid. This information may be helpful for identifying the owner of this client ID and prevent further harm on the network.

Shards are typically 'mirrored' multiple times on the storj network for backup purposes, which means that a 'questionable' shard may end up on more than one farmer. It is important that these shards are removed from each farmer; otherwise these files may end up on a farmer that might distribute the data even more and further harming the network. By using the knowledge gained from the previous steps, we were able to find other farmers, that contain the 'questionable' shard, by checking the *mirrors* collection on the bridge's database. Each key in the *mirrors* collection contains two important fields: the *shardHash* and *contact* fields. The *shardHash* field is used for

finding mirrors of the 'questionable' shard and the *contact* field is for discovering the farmer ID that contains a 'mirrored' 'questionable' shard. Using this led us to discover a key that contained another farmer ID which has 'mirrored' the 'questionable' shard. Taking the discovered farmer ID, we viewed the *contacts* collection in the bridge's database and matched the ID to a key. Once the key was identified, the *address* field displayed the IP address of the discovered farmer and now they can be contacted about the 'questionable' shard. Further metadata samples are shown in Table A.3 of Appendix A which contains information from the conducted tests on our private network.

9. Conclusion & future work

Our work showed that we are indeed able to send unencrypted shards to people's computers that are acting as renters on the Storj network. The implications of the work goes to show that the privacy of data being stored may be compromised when it is finally stored.

However, our work opens the potential for future work. Our work motivates the exploration of forensic carving techniques that may take into account, shard variability, which may become a potential issue in future iterations of this attack. It also motivates the examination of secure architectural choices by peer to peer cloud storage services. Furthermore, future work should attempt to implement techniques for unencrypted shard validation by the renters, as well as a client validation scheme that only allows unmodified clients to join the Storj network.

10. Related work

The proceeding sections review various works relevant to the security and forensic analysis of cloud storage. These works catalyzed the desire and interest for analyzing the security of storj.

10.1. Cloud storage security

Research in vulnerability analysis of cloud computing has been increasing over the past few years; producing a greater understanding of how to identify and categorize their security and privacy flaws. The work by Grobauer et al. (2011) has accomplished this by defining four indicators of cloud-specific vulnerabilities, introducing a security-specific cloud reference architecture, and providing examples of cloud-specific vulnerabilities for each architecture component. Furthermore, various critical security challenges linked to cloud security have been outlined (Ren et al., 2012). Encouraging researchers to further explore the public cloud's challenging security issues.

Through their work, it is possible to highlight the lagging security issues involving cloud security. A perfect example of this is shown through the work done by Xiao and Xiao (2013), where they procured defensive strategies and suggestions to mitigate the challenging issues of cloud vulnerabilities through the approach of separating them into groups.

Additional research encompassing cloud computing security obstacles produced various new concepts and methodologies for improving security in the cloud. One idea is that existing security and privacy solutions must be critically reevaluated with regard to their appropriateness for clouds (Takabi et al., 2010). Another suggested solution, from the work by Chen and Zhao (2012), is that the separation of sensitive data from non-sensitive data is of utmost importance followed by encryption of sensitive information.

Security risks in the correctness of users' data across a distributed cloud has been a major issue. The work by Wang et al. (2012) proposes a solution to the dilemma through their coined scheme: homomorphic token with distributed verification of erasure-coded data.

Other models, such as the Provable Data Protection (PDP) model, are proposed as solutions for the problem of efficiently proving the integrity and validity of data stored at untrusted servers (Erway et al., 2015). Proofs of retrievability (POR) was compared with PDP and they concluded that PDP is the scheme of choice if cloud performance is paramount. However, they continue with stating that POR schemes should be employed in environments where data stored in the cloud is highly-sensitive.

There is great concern over the validity and integrity of data stored on the cloud, especially with sensitive information. Fortunately, unique methods have been developed for mitigating this highly discussed issue. A new efficient variation of the provable data possession (PDP) scheme has been developed and it is called cooperative provable data possession (CPDP) (Zhu et al., 2012). Their new scheme is shown to resist various kinds of attacks even if it is deployed as a public audit service in clouds; effectively preserving the integrity of data in the cloud.

Another proposed method for data integrity verification is an identity-based Remote data integrity checking (ID-based RDIC) scheme to verify that the owner's data is stored correctly (Yu et al., 2017). In order to achieve this, they implemented a key-homomorphic cryptographic primitive with a security model including security against a malicious cloud server and zero knowledge privacy against a third party verifier. Similarly, an efficient public auditing solution, discussed by Yang et al. (2016), shows that they can preserve identity privacy while maintaining identity traceability for cloud data sharing. This concept uses a group manager to help members generate authenticators to protect the identity privacy along with two lists for tracing members who performed modifications on each block.

Using managers as authenticators brings up a similar idea of using third parties as a valid solution for security issues in cloud storage. Implementing a security platform that exists within the cloud storage system will eliminate the security purposes of a third party. File assured deletion (FADE) is a security overlay developed by Tang et al.

(2012) that makes deleted files unrecoverable. FADE operates by using cryptographic keys as a way of achieving access control and assured deletion. Their security overlay was shown to work with Amazon S3 along with presenting FADE's performance-security trade-off. On the other hand, a Trusted Third Party (TTP) is presented as a way to ensure the authentication, integrity, and confidentiality of involved data and communications (Zissis and Lekkas, 2012). Through their proposed TTP using Public Key Infrastructure it was shown to address the identified threats in cloud computing as a solution.

With all of the proposed solutions shown thus far, there still have been exploits that have been discovered in cloud storage services. Online slack space is a cloud storage attack vector that is described in (Mulazzani et al., 2011). This occurs when a cloud storage service is exploited by hiding files in the cloud with unlimited storage space. Attackers with the ability to store unlimited amount of data have easy access to store and/or distribute malicious files to other users of the cloud service.

10.2. Cloud forensics

As cloud computing advances so does the challenge of cloud forensics. Results of Ruan et al. (2013), a survey paper about cloud forensics problems, show that cloud forensics poses significant challenges to digital forensics, and a set of procedures for cloud investigations is needed. Furthermore, the lack of international collaboration and the legal and jurisdictional issues limit access to cloud evidence (Ruan et al., 2011; Zawoad and Hasan, 2013; Damshenas et al., 2012). Through examining the complexity of forensic analysis of cloud computing, Taylor et al. (2011) concludes that the analyst will require a solid understanding of many different technologies and applications. Possible solutions on different phases of digital forensics is proposed in Pichan et al. (2015), producing the result that one of the main challenges is the identification, acquisition, and preservation of data in a cloud environment. The above surveys highlight the difficulties encompassing cloud forensics in a high level perspective and pave the way for examining more specific forms of cloud computing.

One such form of cloud computing is the centralized cloud storage architecture. Chung et al. (2012) studied digital forensic investigations of cloud storage services on PCs and mobile devices concluding that it is necessary and possible to investigate cloud storage services for operating systems other than Windows, such as macOS, iOS, and Android. In addition to investigating for operating systems, data can be acquired from various centralized cloud service providers. Quick and Choo (2013) focused on data collection on three popular public cloud storage products, Dropbox, Google Drive, and Microsoft SkyDrive. The results showed that the downloaded files were identical to the original files. Some of the time stamps of the original files were preserved in the downloaded files, and some were not. Having a framework to follow or tools at your disposal greatly improves a forensic examiners' ability to acquire evidence from the cloud without compromising the data. Secure-Logging-as-a-Service (SecLaas) is a framework that provides logs collected from the open source cloud platform OpenStack to forensic investigators (Zawoad et al., 2013). This framework is beneficial because it protects the integrity of the logs that are crucial evidence in the forensic investigation. Along with that, Federici (2014) presented a tool called Cloud Data Imager, which provides a read only access to files and metadata from Dropbox, Google Drive and MS storage facilities. Another tool called FROST, is an acquisition tool set for the OpenStack cloud platform (Dykstra and Sherman, 2013). This tool was shown to provide trustworthy forensic acquisition of virtual disks, API logs, and guest firewall logs.

Through the use of a framework, forensic investigators can efficiently and safely acquire evidence from cloud networks. Cloud networks are complex, making the difficulty of gathering information all the more challenging. The work by Martini and Choo (2012) brings attention to criminal exploitation of cloud computing and examines forensic frameworks to identify the required changes to conduct successful cloud computing investigations. It proposes an integrated conceptual digital forensic framework, with an emphasis on differences between collection and preservation of forensic data. File storage services, including Dropbox, were analyzed and security improvements were proposed (Mulazzani et al., 2011). From the proposal, it was indicated that cloud storage operators should employ data possession proofs on clients.

On the other side of the spectrum of cloud computing is decentralized cloud networks. Decentralized cloud services present a major forensic challenge in the form of data being widely distributed over many systems, further increasing the difficulty of acquiring information from the cloud. Alenezi et al. (2017) presents their analysis on digital investigations in cloud environments, including both centralized and decentralized clouds. Their result provides a proposal for a framework of technical, legal, and organization factors for digital forensic readiness. Along with that, a traditional digital investigation on a decentralized cloud environment was examined and a suggestion of new solutions and methodologies on decentralized cloud environments was proposed (Birk and Wegener, 2011). A form of decentralized cloud networks is a P2P network in which the work by Liberatore et al. (2010) analyzed two P2P protocols, Gnutella and BitTorrent, and discussed specific interests of forensic investigators. The paper provided principles and techniques for networking investigations, and presented RoundUp, a tool that follows the suggested principles for Gnutella investigations. Research was also performed on BitTorrent Sync for determining the data remnants from the use of the P2P cloud storage (Teing et al., 2017). Their findings showed that artifacts involving installation, uninstallation, log-in log-off and file synchronization, which are valuable to investigators, can be recovered. The Ares P2P network was forensically analyzed by Kolenbrander et al. (2016) with regards to the

distribution of Child Abuse Material (CAM). The paper describes some of the artifacts found on a computer that used Ares for CAM.

10.3. File identification

Identifying files on cloud storage networks is a topic closly related to our findings. The ability to determine a file will mitigate unwanted files and uncover files that may be crucial to a forensic investigation. An online cloud anomaly detection approach was introduced to detect malicious data (Watson et al., 2016). The described approach detected malware with over 90% accuracy, and showed that it was successful in detecting anomalies. In addition, Li et al. (2005) present the concept of determining file types by applying n-gram analysis where they showed that this technique was successfully used to efficiently determine file types, which can be critical to a forensic investigation. Another proposal is a new string search process to improve information retrieval (Beebe and Dietrich, 2007). The described process was designed to more effectively and efficiently search for strings. Furthermore, Garfinkel et al. (2010) explores the use of purpose-built functions and cryptographic hashes of small data blocks for improving the identification and detection of data within files. An algorithm was developed and made it possible to accurately recognize a fragment of a JPEG or MPEG file. Through the use of a multi-tier decision problem that quickly validates or discards byte strings, it was found to be the best method for quickly and accurately carving files based upon their content (Garfinkel, 2007).

Another method of identifying files is through the use of entropy. By utilizing entropy it is possible to determine whether or not a file is in clear-text or encrypted / compressed. If a file is encrypted then the data is secure. However, if a file is in clear-text then the file's content can be seen and / or may contain illegal content. An example of entropy being used for identifying files was by Lyda and Hamrock (2007) where they applied entropy analysis to discover encrypted and packed malware samples. The study found that entropy analysis allowed for the rapid and efficient identification of encrypted and packed malware. Along with that, pattern recognition techniques for fast detection of packed and encrypted malicious executables was implemented (Perdisci et al., 2008). The results showed that their pattern recognition system efficiently and accurately identified packed vs. non-packed executables, so that only packed executables would be sent to a universal unpacker, saving processing time.

References

Alenezi, A., Hussein, R., Hussein, K., Walters, R. and Wills, G. (2017), 'A framework for cloud forensic readiness in organizations'.

Beebe, N. and Dietrich, G. (2007), A new process model for text string searching, *in* 'IFIP International Conference on Digital Forensics', Springer, pp. 179–191.

Beebe, N. L., Maddox, L. A., Liu, L. and Sun, M. (2013), 'Sceadan: using concatenated n-gram vectors for improved file and data type classification', *IEEE Transactions on Information Forensics and Security* **8**(9), 1519–1530.

Birk, D. and Wegener, C. (2011), Technical issues of forensic investigations in cloud computing environments, *in* 'Systematic Approaches to Digital Forensic Engineering (SADFE), 2011 IEEE Sixth International Workshop on', IEEE, pp. 1–10.

Carney, M. and Rogers, M. (2004), 'The trojan made me do it: A first step in statistical based computer forensics event reconstruction', *International Journal of Digital Evidence* **2**(4), 1–11.

Chen, D. and Zhao, H. (2012), Data security and privacy protection issues in cloud computing, *in* 'Computer Science and Electronics Engineering (ICCSEE), 2012 International Conference on', Vol. 1, IEEE, pp. 647–651.

Chung, H., Park, J., Lee, S. and Kang, C. (2012), 'Digital forensic investigation of cloud storage services', *Digital Investigation* **9**(2), 81–95.

Damshenas, M., Dehghantanha, A., Mahmoud, R. and bin Shamsuddin, S. (2012), Forensics investigation challenges in cloud computing environments, *in* 'Cyber Security, Cyber Warfare and Digital Forensic (CyberSec), 2012 International Conference on', IEEE, pp. 190–194.

Dykstra, J. and Sherman, A. T. (2013), 'Design and implementation of frost: Digital forensic tools for the openstack cloud computing platform', *Digital Investigation* **10**, S87–S95.

Erway, C. C., Küpçü, A., Papamanthou, C. and Tamassia, R. (2015), 'Dynamic provable data possession', *ACM Transactions on Information and System Security (TISSEC)* **17**(4), 15.

Federici, C. (2014), 'Cloud data imager: A unified answer to remote acquisition of cloud storage areas', *Digital Investigation* **11**(1), 30–42.

Garfinkel, S. L. (2007), 'Carving contiguous and fragmented files with fast object validation', *Digital Investigation* **4**, 2–12.

Garfinkel, S., Nelson, A., White, D. and Roussev, V. (2010), 'Using purpose-built functions and block hashes to enable small block and sub-file forensics', *Digital Investigation* **7**, S13–S23.

Grobauer, B., Walloschek, T. and Stocker, E. (2011), 'Understanding cloud computing vulnerabilities', *IEEE Security & Privacy* **9**(2), 50–57.

Hall, G. A. et al. (2006), 'Sliding window measurement for file type identification'.

Kolenbrander, F., Le-Khac, N.-A. and Kechadi, M.-T. (2016), Forensic analysis of ares galaxy peer-to-peer network, *in* 'Proceedings of the Conference on Digital Forensics, Security and Law', Association of Digital Forensics, Security and Law, p. 21.

Li, W.-J., Wang, K., Stolfo, S. J. and Herzog, B. (2005), Fileprints: Identifying file types by n-gram analysis, *in* 'Information Assurance Workshop, 2005. IAW'05. Proceedings from the Sixth Annual IEEE SMC', IEEE, pp. 64–71.

Liberatore, M., Erdely, R., Kerle, T., Levine, B. N. and Shields, C. (2010), 'Forensic investigation of peer-to-peer file sharing networks', *Digital Investigation* **7**, S95–S103.

Lyda, R. and Hamrock, J. (2007), 'Using entropy analysis to find encrypted and packed malware', *IEEE Security & Privacy* **5**(2).

Martini, B. and Choo, K.-K. R. (2012), 'An integrated conceptual digital forensic framework for cloud computing', *Digital Investigation* **9**(2), 71–80.

Mulazzani, M., Schrittwieser, S., Leithner, M., Huber, M. and Weippl, E. R. (2011), Dark clouds on the horizon: Using cloud storage as attack vector and online slack space., *in* 'USENIX security symposium', San Francisco, CA, USA, pp. 65–76.

Perdisci, R., Lanzi, A. and Lee, W. (2008), 'Classification of packed executables for accurate computer virus detection', *Pattern recognition letters* **29**(14), 1941–1946.

Pichan, A., Lazarescu, M. and Soh, S. T. (2015), 'Cloud forensics: Technical challenges, solutions and comparative analysis', *Digital Investigation* **13**, 38–57.

Quick, D. and Choo, K.-K. R. (2013), 'Forensic collection of cloud storage data: Does the act of collection result in changes to the data or its metadata?', *Digital Investigation* **10**(3), 266–277.

Ren, K., Wang, C. and Wang, Q. (2012), 'Security challenges for the public cloud', *IEEE Internet Computing* **16**(1), 69–73.
Ruan, K., Carthy, J., Kechadi, T. and Baggili, I. (2013), 'Cloud forensics definitions and critical criteria for cloud forensic capability: An overview of survey results', *Digital Investigation* **10**(1), 34–43.
Ruan, K., Carthy, J., Kechadi, T. and Crosbie, M. (2011), Cloud forensics, *in* 'IFIP International Conference on Digital Forensics', Springer, pp. 35–46.
Takabi, H., Joshi, J. B. and Ahn, G.-J. (2010), 'Security and privacy challenges in cloud computing environments', *IEEE Security & Privacy* **8**(6), 24–31.
Tang, Y., Lee, P. P., Lui, J. C. and Perlman, R. (2012), 'Secure overlay cloud storage with access control and assured deletion', *IEEE Transactions on dependable and secure computing* **9**(6), 903–916.
Taylor, M., Haggerty, J., Gresty, D. and Lamb, D. (2011), 'Forensic investigation of cloud computing systems', *Network Security* **2011**(3), 4–10.
Teing, Y.-Y., Dehghantanha, A., Choo, K.-K. R. and Yang, L. T. (2017), 'Forensic investigation of p2p cloud storage services and backbone for iot networks: Bittorrent sync as a case study', *Computers & Electrical Engineering* **58**, 350–363.
Wang, C., Wang, Q., Ren, K., Cao, N. and Lou, W. (2012), 'Toward secure and dependable storage services in cloud computing', *IEEE transactions on Services Computing* **5**(2), 220–232.
Watson, M. R., Marnerides, A. K., Mauthe, A., Hutchison, D. et al. (2016), 'Malware detection in cloud computing infrastructures', *IEEE Transactions on Dependable and Secure Computing* **13**(2), 192–205.
Xiao, Z. and Xiao, Y. (2013), 'Security and privacy in cloud computing', *IEEE Communications Surveys & Tutorials* **15**(2), 843–859.
Yang, G., Yu, J., Shen, W., Su, Q., Fu, Z. and Hao, R. (2016), 'Enabling public auditing for shared data in cloud storage supporting identity privacy and traceability', *Journal of Systems and Software* **113**, 130–139.
Yu, Y., Au, M. H., Ateniese, G., Huang, X., Susilo, W., Dai, Y. and Min, G. (2017), 'Identity-based remote data integrity checking with perfect data privacy preserving for cloud storage', *IEEE Transactions on Information Forensics and Security* **12**(4), 767–778.
Zawoad, S., Dutta, A. K. and Hasan, R. (2013), Seclaas: secure logging-as-a-service for cloud forensics, *in* 'Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security', ACM, pp. 219–230.
Zawoad, S. and Hasan, R. (2013), 'Cloud forensics: a meta-study of challenges, approaches, and open problems', *arXiv preprint arXiv:1302.6312* .
Zhu, Y., Hu, H., Ahn, G.-J. and Yu, M. (2012), 'Cooperative provable data possession for integrity verification in multicloud storage', *IEEE transactions on parallel and distributed systems* **23**(12), 2231–2244.
Zissis, D. and Lekkas, D. (2012), 'Addressing cloud computing security issues', *Future Generation computer systems* **28**(3), 583–592.

Appendix A. Metadata summary

This appendix provides a table with a subset of metadata gathered from a farmer server and the bridge server on our private network. The metadata shown for the bridge contains a handful of collections from its database and the metadata for the farmer contains its configuration information along with log information from Storj communication messages. The 'Relation' column on all the tables is used to correlate values to one another. For example, if there is a row with one or more relations, then there will be at least one other row, from the tables, for each indicated relation. Additionally, relations spread across all tables. Data shown below is from a test we conducted on the network.

Table A.3 contains information from four collections on the bridge's database. The 'contacts' collection provides helpful information in regards to farmers on the Storj network. Fields such as *_id* and *address* provide a unique identifier and the public IP address for the farmer. Furthermore, the *_id* field has a relation, *A*, to another entry. In this case, the matching entry is the *farmerID* field in the 'exchangereports' collection in table A.3. The 'exchangereports' collection contains information about data transfers of shards on the network. Information such as the hash of a shard, email address of a client user, ID of a farmer, exchange time, and the exchange message can be extracted and provide great insight as to what, where, and when a shard is being transfered. The value of the hash is helpful for correlating other entries in collections and for verifying the data integrity of shards on a farmer. Within the 'mirrors' collection, the hash of the shard, the starting storage time of the shard, the source shard's renter ID (can be a client or farmer), the destination shard's farmer ID, and the data size of the shard can be found. This collection can aid a forensic examiner by verifying shard hashes and determining other farmers that have shards with the same hash. Under the 'users' collection, user information can be found. An email address, hashed password, byte transfer rates, and user creation time are maintained within this table.

Configuration information about a farmer is shown in table A.4. The table contains the file location for the configuration file. A configuration file will contain helpful information such as the IP address of the bridge, a *seedList* that contains IP addresses for 'always known' renters on the network (to gain access to the Storj network), and port information. Having the bridge IP address is extremely valuable and this is how an analyst can acquire the IP address from a farmer.

Lastly, table A.5 contains communication messages between the farmer who owns the log file and another farmer or the bridge. The table contains the location for the message log file. Various message types are shown in the table and messages for the client shard uploading process is provided along with the client shard downloading process. The client shard uploading and downloading process has the starting message of 'received valid message from' and an ending message of 'Shard ... completed hash'. The intermediate messages are slightly different but are understood based upon the message name. Most of the entries in the table contain information regarding the hash value of the transfered shard and the farmer IP address of the originating message. In this case, we will focus on the 'received valid message from' and 'handling storage ...' messages. The valuable data from a 'received valid message from' entry are the farmer's ID and IP address. On the other hand, the 'handling storage ...' message contains the shard hash and the requesting farmer ID.

Table A.3: Shard metadata contained on the bridge server

Bridge Server Metadata			
Collections	**Fields in key**	**Field value**	**Relation**
contacts	_id	"5a2f43355526137739603667 2418f765b51f0de9"	A
	lastSeen	ISODate("2017-12-24T20:52:19.192Z")	
	port	9001	C
	address	x.x.x.x (Blinded for security)	
exchangereports	_id	ObjectId("5a400e26a30a597ecb68e1da")	
	dataHash	"3ebf21a993b723e638a283035d2dc572f16b9f56"	F
	reporterId	"storj-test@trash-mail.com"	G
	farmerId	"5a2f43355526137739603667 2418f765b51f0de9"	A
	clientId	"storj-test@trash-mail.com"	G
	exchangeStart	ISODate("2017-12-24T20:29:25.977Z")	I
	exchangeEnd	ISODate("2017-12-24T20:29:25.996Z")	J
	exchangeResultCode	1000	
	exchangeResultMessage	"SHARD_UPLOADED"	L
	created	ISODate("2017-12-24T20:29:26.004Z")	M
mirrors	_id	ObjectId("5a400e259cf6997ef41f5420")	
	shardHash	"3ebf21a993b723e638a283035d2dc572f16b9f56"	F
	contact	"f33e0bfe6eaa263bf2eb08b9d919c80ac5c30157"	
	store_end	1521923364847.0	
	store_begin	1514147364847.0	
	renter_id	"337472da3068fa05d415262baf4df5bada8aefdc"	R
	payment_storage_price	0	
	payment_download_price	0	
	payment_destination	"0x4780cA5a6E8cA5a950390f2bb9e7Fa11822A46b9"	
	farmer_id	"f33e0bfe6eaa263bf2eb08b9d919c80ac5c30157"	
	data_size	64	
	data_hash	"3ebf21a993b723e638a283035d2dc572f16b9f56"	F
	audit_count	4	
	isEstablished	true	
users	_id	"storj-test@trash-mail.com"	
	hashPass	"5cf2..." (Blinded for security)	
	referralPartner	null	
	bytesDownloaded	(3 values)	
	lastMonthBytes	0	
	lastDayBytes	0	
	lastHourBytes	0	
	bytesUploaded	(3 values)	
	lastMonthBytes	4194368	
	lastDayBytes	64	
	lastHourBytes	64	
	isFreeTier	true	
	activated	true	
	created	ISODate("2017-09-03T18:31:55.454Z")	
	pendingHashPass	null	
	uuid	"7d18d510-1e80-4c9f-991e-d0e486c76601"	

Table A.4: Shard metadata contained on the a farmer server's config

Farmer Server Metedata

File: /home/storjtest/.config/storjshare/configs/5a2f43355526137739603667 2418f765b51f0de9.json

Configuration setting	Setting value	Relation
paymentAddress	"0x4780cA5a6E8cA5a950390f2bb9e7Fa11822A46b9"	
bridgeURI	"http://192.168.1.117:6382"	
seedList	(3 elements)	
	"storj://192.168.1.117:4000/337472da3068fa05d415262baf4df5bada8aefdc"	R
	"storj://192.168.1.117:4001/b2e1173bf733aeaacf79bf73a5a65bc5a912d923"	T
	"storj://192.168.1.117:4002/b78c3ad6007e316e38a2bab0d567a617f6b98fe6"	
rpcPort	9001	C
tunnelGatewayRange	(2 elements)	
min	4001	U
max	4003	

Table A.5: Shard metadata contained within a farmer server's log file

Farmer Server Metedata

File: /home/storjtest/.config/storjshare/shares/<farmernodeid>/contracts.db/<*.log||*.ldb>

Log message	Message	Timestamp	Relation
received valid message from	"address":"x.x.x.x" (blinded for security) "port":4000 "nodeID":"337472da3068fa05d415262baf4df5bada8aefdc" "lastSeen":1514147194758	2017-12-24T20:29:25.553Z	R
Offer accepted		2017-12-24T20:29:25.590Z	
handling storage consignment request from	337472da3068fa05d415262baf4df5bada8aefdc hash 3ebf21a993b723e638a283035d2dc572f16b9f56	2017-12-24T20:29:25.849Z	R F
authorizing upload data channel for	337472da3068fa05d415262baf4df5bada8aefdc	2017-12-24T20:29:25.867Z	I, J, R
Shard upload complete hash	3ebf21a993b723e638a283035d2dc572f16b9f56	2017-12-24T20:29:25.987Z	F, I, J, L, M
received valid message from	"address":"x.x.x.x" (blinded for security) "port":4001 "nodeID":"b2e1173bf733aeaacf79bf73a5a65bc5a912d923" "lastSeen":1514147204800	2017-12-24T20:29:26.148Z	U T
handling storage retrieve request from	b2e1173bf733aeaacf79bf73a5a65bc5a912d923 hash 3ebf21a993b723e638a283035d2dc572f16b9f56	2017-12-24T20:29:26.151Z	T F
authorizing download data channel for	b2e1173bf733aeaacf79bf73a5a65bc5a912d923	2017-12-24T20:29:26.152Z	T
Shard download completed hash	3ebf21a993b723e638a283035d2dc572f16b9f56	2017-12-24T20:29:26.278Z	F