Torrage: A Secure BitTorrent based Peer-to-Peer Distributed Storage System

Debarghya Das DD367@CORNELL.EDU
Cornell University, Ithaca, NY 14850

Abstract

Most cloud storage platforms today offer a centralized service to store your files, typically with a freemium model - free up to a certain limit. A centralized service is inherently insecure and unreliable because your data is both accessible and dependent on a single service. Current US government regulation, for example, allows for legal invasion into a users privacy. The distributed peer-to-peer model has been known to provided a source of anonymity and security to web browsing - like Tor and Bittorent. Peer to peer storage has always come with a set of constraints that made it hard to market as a viable competitor to cheap centralized alternatives by well-funded private companies like Dropbox, Google Drive, and Microsoft SkyDrive. We present Torrage, , a secure p2p storage system that proposes to serve as a potential viable alternative.
The report is divided into 3 main sections - a primer on BitTorrent, an explanation of the system design of the entire system, and details on our current implementation.

1. BitTorrent

BitTorrent was initially designed to be a protocol for distributing (typically large) files. It identifies content by URL and is designed to integrate seamlessly with the web. Its advantage over plain HTTP is that when multiple downloads of the same file happen concurrently, the downloaders upload to each other, making it possible for the file source to support very large numbers of downloaders with only a modest increase in its load. [1]
In 2009, BitTorrent was estimated to have around 43 - 70% of all Internet traffic [2] and the number of monthly users as of January 2012 is estimated to be 250 million [3].
A user can create a small torrent file for a large file or many files which contains metadata such as the filename, and cryptographic hashes of fixed size chunks of the file. When a user obtains a torrent file, traditionally, they would contact a Tracker - centralized servers that informed peers of the seeders of a certain torrent. Owing to security concerns, torrents now are mostly trackerless [4]. In trackerless torrents, torrents typically ping a centralized server to get the IP address and port number of a certain node, and each peer for all torrents serve as part of a Distributed Hash Table (DHT), with the Kademlia protocol over UDP. Depending on the "info hash" identifier of the torrent you are looking for, each peer returns the address of a peer that is guaranteed to be *closer* to knowing what other peers are downloading the torrent. Finally, it receives this list and begins trying to handshake with them, and acquiring chunks of the file randomly, typically based on a *most rare first* heuristic. The size of torrent files themselves is proportional to the size of the file it is created from, but they are on the order of 10,000 times smaller. The torrent files for around 1, TB of data can be expected to be around 50MB, which is only a 0.005% space overhead.

[0] Not to be confused with https://torrentfreak.com/torrage-worlds-first-torrent-storage-service-090806/, an unrelated torrent hosting service which unforunately had the same name

[1] http://www.bittorrent.org/beps/bep_0003.html
[2] http://torrentfreak.com/bittorrent-the-one-third-of-all-internet-traffic-myth/
[3] http://www.bittorrent.com/intl/es/company/about/ces_2012_150m_users
[4] Trackers were easy centralized targets for takedown when transmission of illegal content was involved

2. System Design

2.1. Motivation

Cloud storage is a very important space in the industry, with estimates reporting a market size of over $100 billion [5]. However, there are several issues plaguing this space:

1. **Security and Privacy** Having centralized services leaves you without anonymity, and gives organizations access to all your data. Further, typically Web 2.0 cloud storage applications transmit password credentials across the wire, and store encrypted salted passwords on their servers, both of which can be compromised. [6] [7]. In fact, recently the popular cloud storage platform Dropbox reported 7 million stolen passwords [8] and Apple's iCloud leaked nude celebrity pictures [9].

2. **Cost and Space restrictions** Although the price of cloud storage has been in steady decline, the services usually work on a freemium model where they grant a certain amount of space (2 - 10GB) for free, and require periodic subscription fees for more space.

3. **Speed** While not a large motivating factor, we can seek to improve upon the speed of existing cloud storage services, especially for large files (e.g. movies).

4. **Popularity** Cloud storage in 2013 had 2.4 billion users and is expected to grow to 3.6 billion users by 2018 [10].

2.2. Goals

The key goals of our p2p distributed storage system are:

1. **Security and Privacy** One of the primary motivating factors of a p2p distributed storage system was higher security. One user's files should not be accessible by MITM attacks on the wire, and will be transferred with complete encryption over the wire. The files should be sharded so that even in the unlikely case of tapping, the data will be corrupt and meaningless at one given node. It should be encrypted with a key only possessed by the owner(s) of the file.

2. **Fault Tolerance** Even if certain peers fail or are down, a users files should be readily accessible. This involves replication of file shards.

3. **Availability** Relying entirely on peers can be dangerous because of the increased possibility of downtime. We attempt to design a system that can provide certain availability guarantees.

4. **Practicality** Each peer to use the system must be forced to allocate some of their resources for the public good. As a consequence of high down time and replication, these resources have to be sufficient to support a practical system.

5. **Speed** When an end user attempts to access their files, he should be able to leverage multiple peers possessing his content to get faster download speeds than a centralized service would provide.

6. **Scalability** When new nodes are added or nodes removed, the system should be smart about reallocating resources.

2.3. System

We explain how our system works by going through each of the steps of a file upload and recovery chronologically:

1. **Encryption** A user has a file f which he wishes to store on the cloud. He possesses a username and a (strong) password combination as credentials. We use the password with **PBKDF2** (Password based key derivation function) and a cryptographic salt (in our case, we use **bcrypt** s to generate a strong 256 bit key. We currently assume that one's files are meant to be read only by themselves, and therefore we use symmetric encryption using AES-256 and our salt to encrypt f *locally* to $E(f)$. [11]

2. **Distribution** We split $E(f)$ into n more or less equal parts, $E(f)_1, E(f)_2, ..., E(f)_n$, using a modulus based scheme (explained later), and look for peers in the system, which may be our own devices as well. Each peer allocates a certain portion of

[5] https://www.linkedin.com/pulse/article/20130729184037-295895-cloud-computing-crosses-the-100-billion-milestone

[6] http://www.thoughtcrime.org/software/sslstrip/

[7] http://en.wikipedia.org/wiki/Rainbow_table

[8] http://arstechnica.com/security/2014/10/7-million-dropbox-usernamepassword-pairs-apparently-leaked/

[9] http://www.theguardian.com/technology/2014/sep/01/naked-celebrity-hack-icloud-backup-jennifer-lawrence

[10] http://www.statista.com/statistics/321215/global-consumer-cloud-computing-users/

[11] We ensure our encrypted file $E(f)$ contains our randomly generated salt.

their hard disk space proportional to the amount they use. The details of this are yet to be experimented with, but the idea is to provide monetary incentive for allocative more than this ratio, and to charge for less than this ratio. For the purpose of this paper, let's assume 100% uptime. At the very least, we can expect other devices owned by the same user to be up. [12] First, we create torrent files for each of n parts, let's call them $T(E(f)_k)$. We establish simple connections with the relevant peers for each of our parts, and transfer the relevant torrent to our peers. If they have enough space to store the part, they begin leeching, and you begin seeding $E(f)_k$ to all the peers you are distributing it to through the standard BitTorrent protocol, and then close the connection. We have yet to decide exact protocols for file updates, removes and adds. After this, the small torrent files are stored on the centralized server, guarded only by a username login. The password does not travel over the wire.

3. **Recovery** The primary purpose of a cloud storage system is as a universally accessible, available backup of your data. Suppose our user wanted to access his file f from a different machine. Firstly, he uses his username to retrieve the torrent files $T(E(f)_k)$ for all $k \leq n$. Then, he begins leeching for his files. [13] The user leeches his file parts until all of them are available to him. After this, he assembles them into $E(f)$, and again, uses the password only *locally* to decrypt it to f.

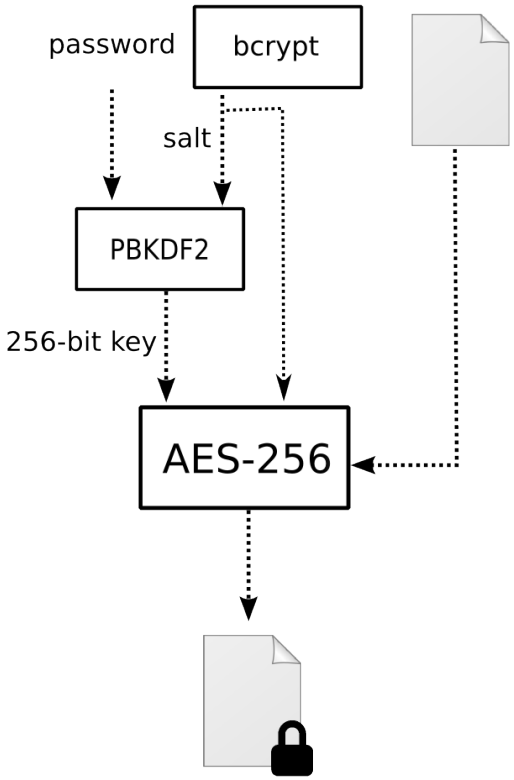

Figure 1. The architecture of the encryption step of the storage process.

2.4. Advantages and Disadvantages

Advantages to our proposed solution:

1. **Very Secure** With the use symmetric encryption with PBKDF2 and AES-256, we ensure that encryption/decryption happen completely on the client. This means that no other service stores your password hashes, preventing recent password leaks from cloud services like Dropbox. Because the content is encrypted client side before being transferred over the wire, it is also not vulnerable to naive MITM attacks nor is it vulnerable to SSL vulnerabilities or SSL evasion tactics like SSLstrip. A user's torrent files are stored on a centralized server and accessible with only a username, providing another security layer. The password does not travel over the wire, and is only used locally for decryption.
The use of PBKDF2 and AES-256 mean that even in the case that a malicious attacker gets access to a user's torrents, and is able to download and assemble the encrypted file, they will not be able to use simply dictionary attacks to decrypt it if we enforce the use of a strong password. Although not implemented, we would ideally use Two Factor Authentication (TFA) as an additional security measure.

2. **Privacy** A peer will only ever have shards of a

[12] In practicality, we could maintain the security benefits while adding a centralized server for storing shards and guaranteeing high uptime.

[13] Note, that because username protection is just a thin layer of protection, even if the server implements correct encryption over the wire with SSL and throttles multiple failed attempts, a good hacker could ostensibly get hold of the torrent files. However, even in this case, even if he obtained $E(f)$, he would not be able to break AES-256 + PBKDF2 with a simple dictionary attack if the password had certain security constraints.

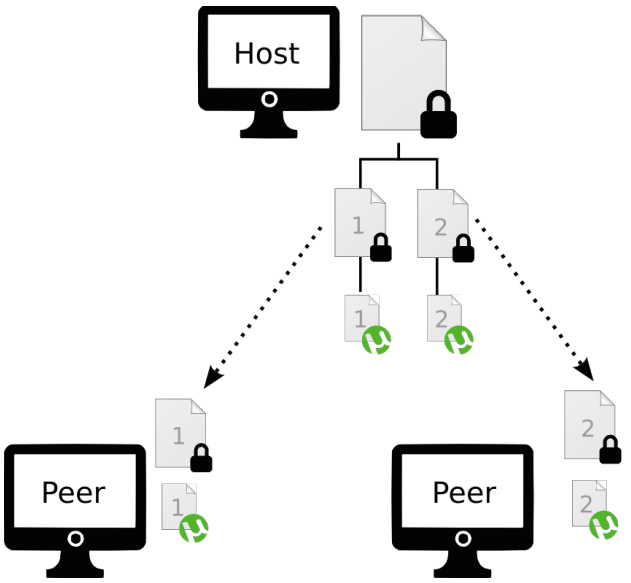

Figure 2. The architecture of the distribution step of the storage process. First, the torrent files are transferred to the peers, and they begin to leech parts of the encrypted file.

user's files, which are encrypted until it leaves the user's machine. Because the content is only ever decrypted on the user's machine, there are no simple packet inspecting rules that can detect the nature of the content.

3. **Faster** Our speed is proportional to the replication factor and inversely proportional to the downtime. In most cases, we should be able to guarantee faster download speeds for larger files. In addition, we could potentially increase the replication factor for larger files to maintain smaller download times for them, and use BitTorrent's support of low-latency video streaming for large video files. If we store data across our own devices that we can guarantee uptime for, we can multiply speeds by the number of devices we have.

4. **No space restrictions** Because in the best case, no data besides torrent files will need to be stored on a server, space is only bound by your device capacity and how much space you want to donate.

5. **Potentially free** Again, because the servers don't store any data, we can offer this service free of cost.

In our proposed solution, we do not include any guarantees regarding scalability, fault-tolerance, availability and practicality. It is not entirely in the scope of this class. All these factors are related, and we are confident that at worst case, our system architecture will allow us to use centralized servers securely to boost the performance of the p2p system while maintaining its security and speed advantages. We could also enforce uptime constraints on our peers or provide monetary incentives for more space. This should guarantee at least the performance of existing system.

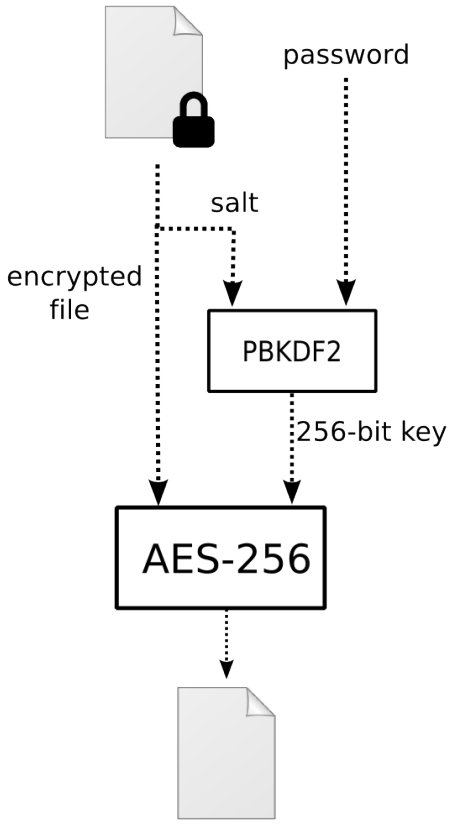

Figure 3. The architecture of the recovery step of the storage process. First the torrent files are retrieved from the centralized server with just the username credential, downloaded, combined, and then decrypted with a password locally.

2.5. Potential Problems

In general, there are several concerns with the high level idea of Torrage:

1. **Redundant Space Use** In a real world scenario, would a Torrage user be willing to give to the community at least as much hard disk space as they themselves are using on Torrage? In the case where we cannot guarantee uptime for our

peers, Torrage would necessitate storage of encrypted files on a centralized server.

2. **Scalability** What protocols would we use for peer load balancing with the dynamic addition and removal of new nodes to the system? Would the load balancing network overhead be too much for the system to handle?

3. **Adding, Removing and modifying files** How would Torrage deal with modifying a large text file at a localized spot? Surely retransmission of the entire file would require too much overhead. Adding files seems simple enough, but what about the protocol for remove encrypted shards when files are removed.

4. **Load-balancing** The difficult algorithmic problem here is - how do we come up with a fast way to load balance all the replicated shards of all the peers across all the other peers on the network, while maintaining the limits that each peer is willing to contribute?

5. **Sharing items** How do we deal with sharing files in groups? How do we deal with making files public?

2.6. Future Work

To add to this base system design, we suggest the following features:

1. **Centralized Server Support** To counter the constraints of peer uptime, amount of space the community is willing to share, and more, we can always use a centralized server to store file shards to boost the performance of the system without compromising on security. We can impose restrictions on how many shards of a file can belong on the server and we can maintain client side passwords to maintain security.

2. **Password changes** Since files are encrypted with a password, what would the procedure for password changes and password losses be. Something to think about.

3. **Redundancy Removal** If multiple people are keeping backups of the same file, how can we, if at all, preserve one copy of the file maintaining the current security benefits?

4. **File Sharing** When somebody wants to share a file, how do we deal with it? Can we incorporate some form of asymmetric encryption for this case?

5. **Compression** How much of an additional speedup can we get by compressing our encrypted files before sending them over the wire?

3. Submission

3.1. Outline

The code consists of 4 modular units tied together by a sample test script -

1. `endecrypt` Encrypts and decrypts files securely. We obtained open-source versions of AES-256, PBKDF2, SHA hash functions [14] and bcrypt (also known as crypt_blowfish) [15].

2. `split` Split the resultant encrypted files modularly into chunks.

3. `join` Join encrypted files encrypted by `split` properly.

4. `torrage` BitTorrent client responsible for network traffic. We model our code around an online specification that encompasses a major portion of the actual BitTorrent protocol [16]. We also use `ctorrent` [17], an open source implementation of

These compiled binaries from these files are then by a master script that demonstrates it's functionality.

3.2. Features

Completely a fully-featured working version of Torrage in C working alone in a month, and disrupting a $100 billion industry is beyond the scope of this course. However, my submission can do the following:

1. Implement complete industry level security with implementations of top of the line modern cryptography.

2. Iterative distribution of multiple files.

3. Distribution onto n peers on different machines over the network.

4. Retrieval and reassembly of the encrypted sharded chunks over the network from the peers successfully, provided the correct credentials.

[14] https://www.openssl.org/
[15] http://www.openwall.com/crypt/
[16] http://www.cs.swarthmore.edu/~aviv/classes/f12/cs43/labs/lab5/lab5.pdf
[17] http://ctorrent.sourceforge.net/

5. Fault Tolerance - if we redundantly shard, the peers can go down, or we can go offline and resume recovery.

Some portions of the system we notably omitted due to time constraints and the jurisdiction of the course:

1. Proper abstraction and usability. Most of the functionality is in the form of automated scripts.

2. A centralized server for storing and distributing the actual torrent files. Currently, we manually transfer torrent files from machine to machine.

3. Implementation of a distributed hash table or a tracker that tells Torrage clients where to look for the peers. We currently manually provide this information.

4. Uptime guarantees for the peers. While peers may go up and down and our process will not crash, we do not have any guarantees that all the shards will be available if some peers shut down permanently.

5. We do not deal with multiple seeders and one leecher for a torrent.

3.3. Functionality

We've tried and tested distribution onto multiple computers, but we've written a test suite that demonstrates the correct functionality of Torrage. Let's go over it step by step.

1. **Initial State** In figure 4, we see the initial state of our demonstration. Although we could work with machines on the cloud, for testing across machines, we decide to emulate machines by using sandboxed folders. Here we see 4 machines - `host1` represents a user's primary device, `host2` his backup device, and the two peers represent other willing donors of space on the network. In the initial state

2. **Encryption, Sharding and Creating Torrent Files** In figure 5, we see the state of the system post-encryption, and torrent creation. We securely encrypt with AES-256 encryption using a password and PBKDF2 with secure random bcrypt salts. After this, we split the encrypted file into n chunks [18]. Then, we use `ctorrent` to generate torrents for these chunks.

[18] In our demonstrations and diagrams, we typically use 2 chunks for simplicity

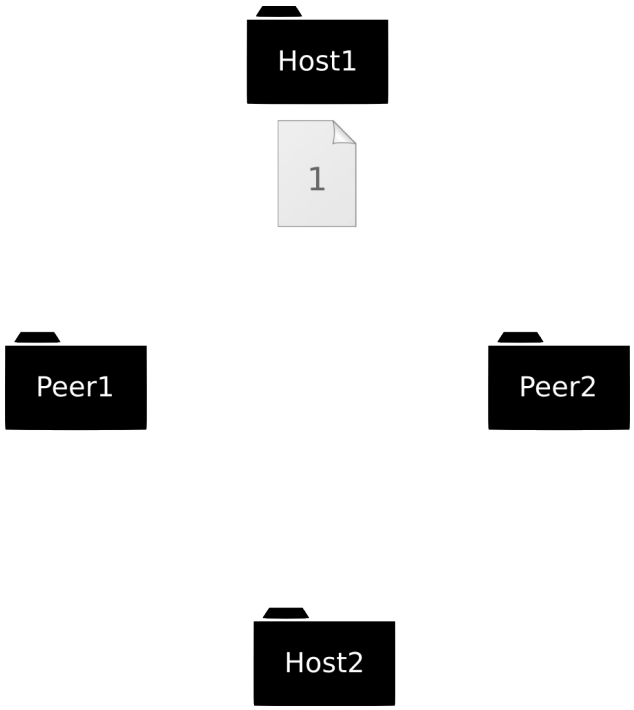

Figure 4. Step 1: The Initial State

3. **Artifical Torrent file distribution** In figure 6, we show our torrent distribution process. In the actual system, a centralized server is in charge of choosing peers, and acquiring and distributing torrents. [19]. Instead, we simulate the distribution by copying the files manually on a filesystem to the peers.

4. **Distribution** In figure 7, we see the distribution process. [20] In our demonstration, we listen on two different ports on localhost from `Host1`, and then the peers bind to different ports, also on localhost and attempt to connect to the `Host1`. Again, because we do not use DHTs, we manually supply the correct addresses to find seeders on. The encrypted file chunks are transmitted over the network to the peers.

5. **Artificial Torrent file distribution to second device** We now want to simulate the recovery phase. Suppose a user has corrupted his `Host1`

[19] Note, that torrent files are small and do not carry much overhead or contain sensitive data. Therefore the existence of a centralized server is inexpensive. We could also use a distributed hash table for this like the actual BitTorrent protocol does

[20] Recall that torrents contain metadata and verification data to allow fast, restart-able downloads, in non-linear order from various different sources.

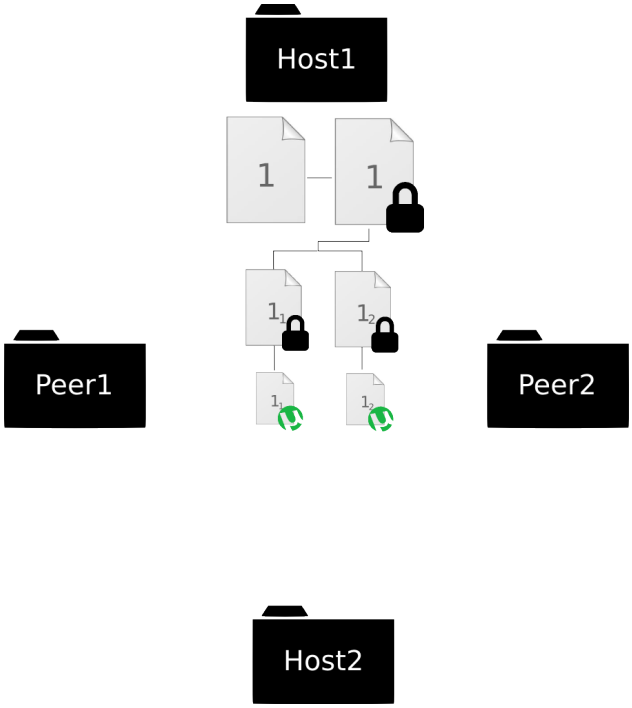

Figure 5. Step 2: Encryption, Sharding and Creating Torrent Files

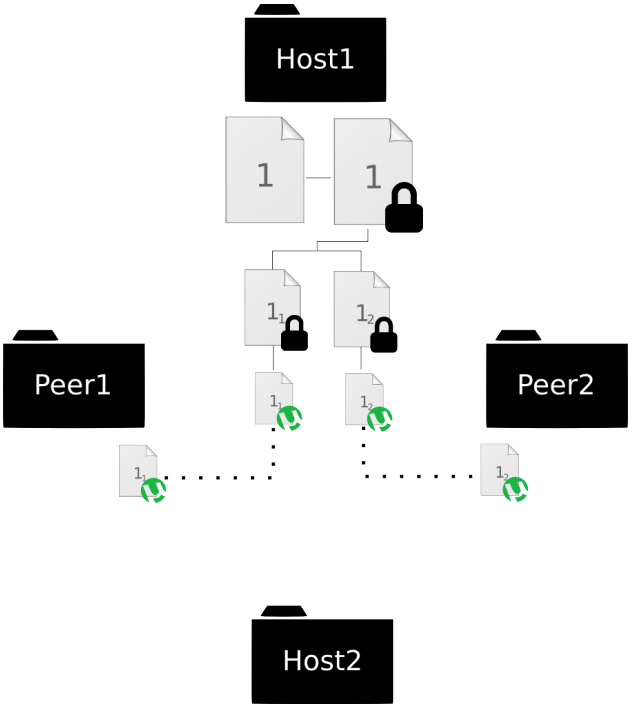

Figure 6. Step 3: Artificial distribution of torrent files

disk, and wants to recover his content from the cloud. As shown in figure 8, we separate `Host1` for the system and first artificially copy the torrent files for the encrypted parts to `Host2` (which would normally be retrieved from the server).

6. **Download of encrypted shards** In figure 9, we show the downloading of the shards through the torrents to the new host device, `Host2`, completely independent of the state of `Host1`.

7. **Recovery** Finally, we see in 10, the end user merges the encrypted files into one piece in accordance with the obfuscated splitting algorithm and decrypts the result with his password to obtain the original file again.

We wrote a script `master_test.py` in Python (for ease of writing) that executes each of these steps in succession, and compares the `SHA-256` hashes of the initial file and the recovered file to ensure successful encryption and decryption. While it is not feasible to ensure perfect security without penetration testing, we assure that the encrypted files are indeed unusable at every stage we desire it to be. We allow an option in the script to allow us to enter passwords ourselves, and another to use the same default password, for performance testing.

3.4. Performance

3.4.1. Security Performance

Torrage works extremely well on all our security performance benchmarks. It works correctly, scales as expected, and is fast enough to make the entire system performance dependent on the network performance

1. **Adequate password verification slowness** We show in figure 11 that we maintain a 0.4s time for key generation, independent of the size of the file. This is just fast enough to not be a UX concern for the user, but slow enough to mitigate attacks. We see, we can arbitrarily tune this time, in figure 12 , by modifying the number of iterations the PBKDF2 key is rehashed. We choose 10^5 to obtain our 0.4s time.

2. **Fast encryption and decryption** We observe a fairly consistent encryption and decryption speed of around 50-60 MBps. This can further be sped up with parallelization. Currently it takes about 15 seconds to encrypt and another 15 seconds to decrypt a full length high quality feature film.

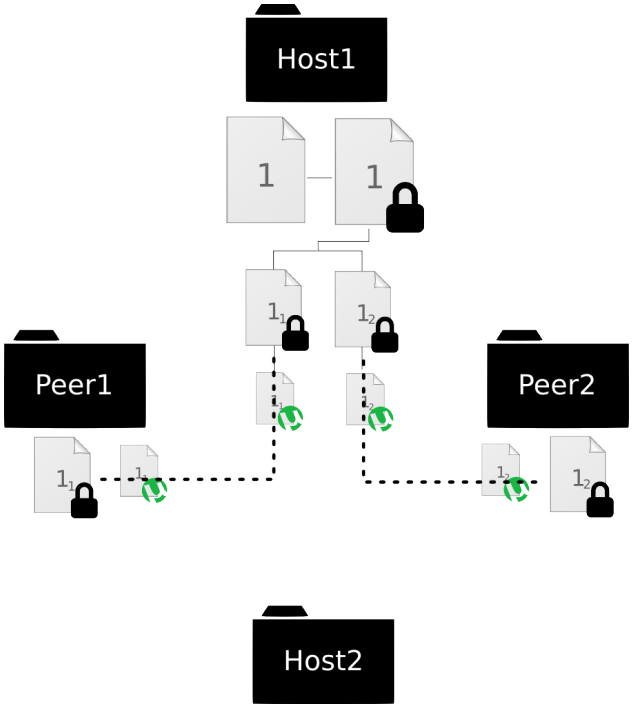

Figure 7. Step 4: Distribution

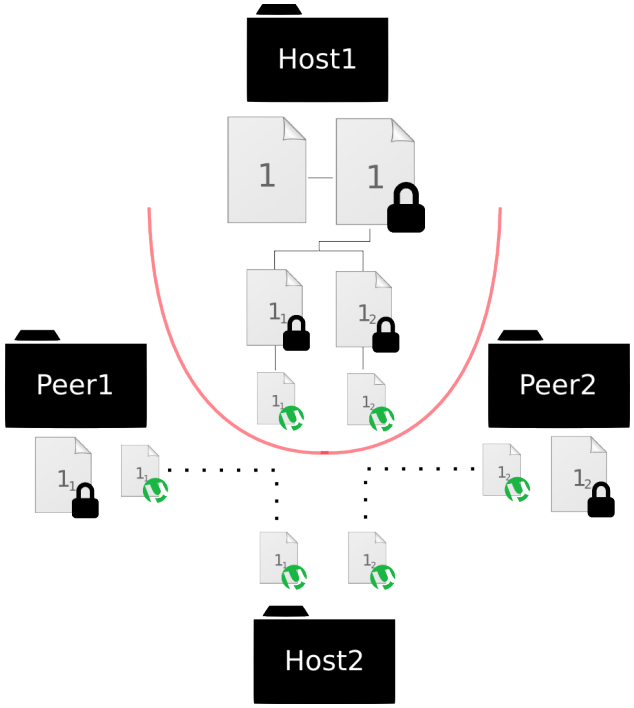

Figure 8. Step 5: Artificial distribution of torrents to the second device

Our performance for encrypting and decrypting files is shown in figure 13.

3. **Fast splitting and joining** Our modular scheme for splitting and joining files runs at about 200 MBps. This can split a feature film in about 3.5s. Our performance for splitting and joining files is shown in figure 15.

3.4.2. Network Performance

Our BitTorrent stack successfully does 1 seeder - multiple leechers. While the leechers can leech from each other and be called peers, the algorithm to make this work efficiently and although it doesn't crash, peers often block each other. We have no implemented multiple seeders to one leecher.

We tested one seeder to 1 and 2 leechers across two different machines (not over localhost). Although this speed is highly network dependent and constrained by the implementation, we simulated it to get an idea of transfer time across a fairly high speed network. In reality, with a complete implementation, the BitTorrent protocol can get speed ups proportional to the the number of seeders. On average, for 1-seeder 1-leecher, we generally achieved a speed of 3 MBps, and for 1-seeder 2-leecher, this increased to an effective speed of 4.5 - 5 MBps [21].

3.4.3. Overall performance

For our current implementation, speed remains constant with varying file size more or less. Time is linear in file size, as one might expect. For a 135MB file, the encryption process takes about 3.5 seconds on the client, and 42 seconds on the network, which is a 3MBps transfer speed overall. This is fairly performant. More importantly, this is network bottlenecked.

3.5. Security

Here are some of the details of the security measures used:

1. **bcrypt salts** We randomly generate a crypt salt, known to be one of the most secure, and we use the highest security setting, 31, for this process.

2. **PBKDF2** PBKDF2 ensures the secure generation of keys from a password, given a salt. To pre-

[21] We download twice the amount from 2 leechers from the same peer "parallely"

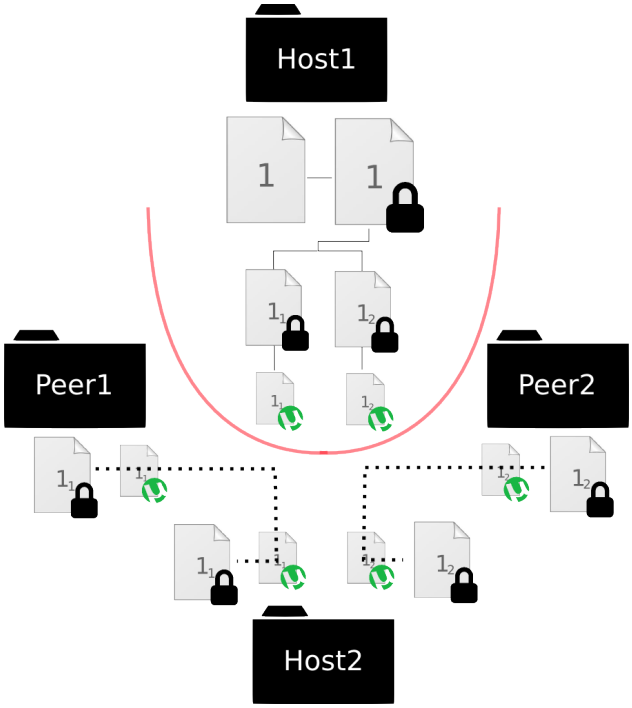

Figure 9. Step 6: Downloading the encrypted shards

vent against attacks with dictionary and rainbow table, we intentionally slowing down the process to 0.5s by hashing our hash for 10^5 iterations.

3. **AES-256** AES-256 is extremely secure and unbroken to date. It uses 256 bit keys and a salt and operates on 128-bit blocks. We use the same bcrypt salt we use for generating our key with PBKDF2..

4. **Obscure sharding** For example, suppose a file is split into 1000 chunks, and we require 10 shards. The first chunk is placed in the first file, the second int he second, and the eleventh back in the first. This process adds another layer of obscurity, wherein the penetrator cannot obtain much relevant information about a file from a single shard, even if he somehow manages to decrypt it.

5. **Air-gapped Password** The password credentials never leave the local machine and this eliminates all the potential threats, despite SSL, over the wire.

6. **BitTorrent based verification** BitTorrent is inherently secure when it comes to file transfer. If a malicious peer tries to feed you bad data, it'll cause a hash mismatch and we reject the transfer.

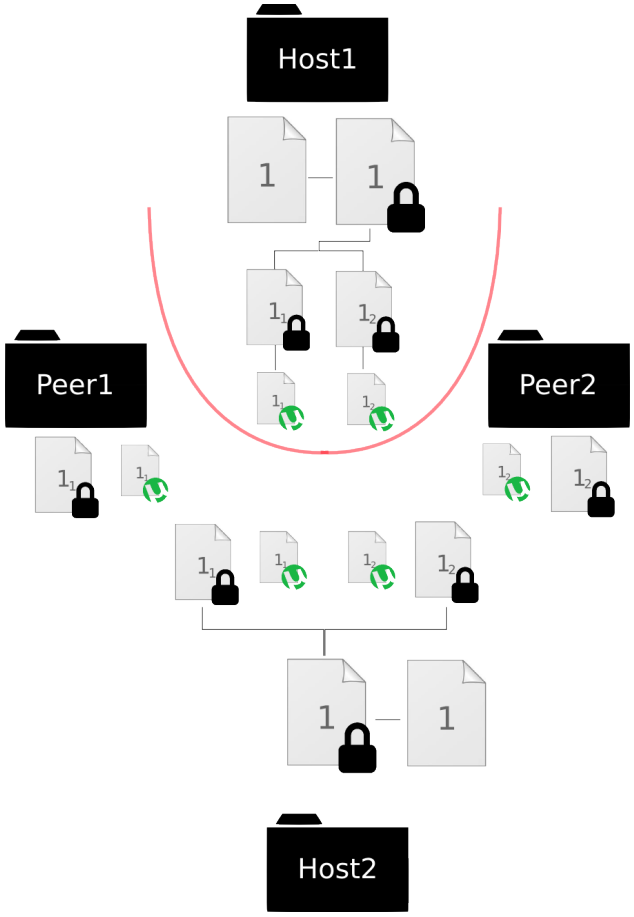

Figure 10. Step 7: Recovering the original file

4. Conclusion

In this paper, we demonstrated a more secure, faster, cheaper and space-unlimited cloud storage system. In addition, we implemented a working, performant proof-of-concept of this system that provided extremely secure file storage and retrieval that is network bottle-necked. We used industry-strength open-source cryptography and worked around a well-known and proven BitTorrent protocol to create a new, functional, and tested system.

t

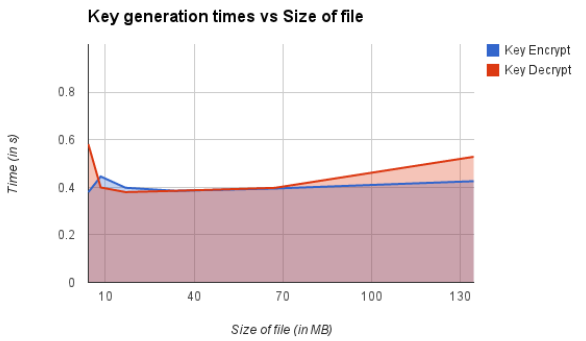

Figure 11. The change in time it takes to generate a key with the size of the file.

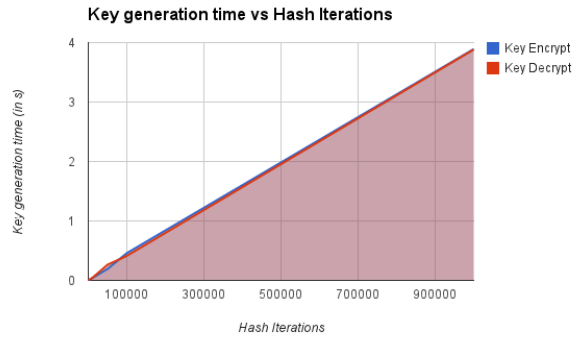

Figure 12. The change in time it takes to generate keys, with the increase in hash iterations in PBKDF2.

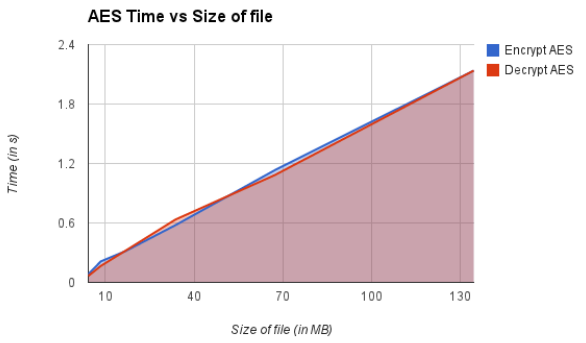

Figure 13. The change of the time it takes to encrypt and decrypt a file with AES with increase in file size.

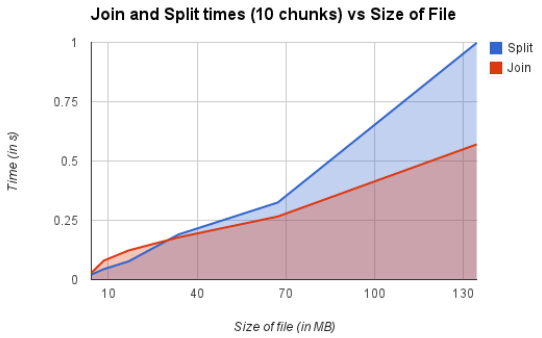

Figure 14. The change in the join and split times of files with increase in file size.

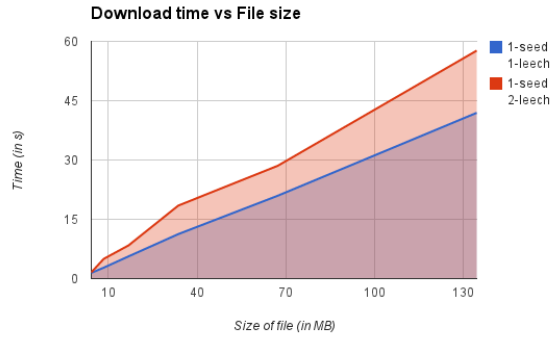

Figure 15. The change in time to transfer files as a function of file size.

Figure 16. The time taken for an end-to-end test, with 2 chunks over the local network and verification of correctness.