Chasm: Fault-Tolerant, Information-Theoretic Secure Cloud Backup using Secret Sharing

Alex Grinman, Akshay Ravikumar, Julian Fuchs, Kevin Li
{agrinman, akshayr, jfuchs, kmli}@mit.edu

https://github.com/agrinman/chasm

5/11/2016

Abstract

Most "secure" cloud backup software encrypts a user's data under a symmetric key that either they store on their computer or backup elsewhere. If a user loses the key when their computer crashes, their encrytped, backed-up data is unrecoverable! Some systems deal with this problem by generating a symmetric key that is dervied from a password the user is asked to memorize. A password sacrifices entropy and ultimately weakens the security of the system. The cost of real security in these types of systems is fault-tolerance, which is the underlying motivation for backing up to the cloud. In this paper we present **Chasm**, a fault-tolerant secure cloud backup solution based on Shamir's k-out-of-n secret-sharing scheme.

1 Motivation

There are many commericial backup solutions that claim to provide confidentiallity and integrity of user data. Some examples include Mozy, Carbonite, Crashplan, and Backblaze. All of these products essentially work the same way: data is encrytped using symmetric encryption along with an messege authentication code for integrity and sent to the storage service's cloud data store. Unfortunately, this mechanism of encrypting the data does not solve any problem, it simply concentrates the problem in a short symmetric key. The maintainence of this symmetric key is central to the fundamental security of the system.

The threat model that most of these existing solutions operate under is that the adversary is a "hacker" who breaks into the storage service and steals data. Hence, in this adversarial model, the hacker will not be able to decrypt the data without the key.

However, we find three problems with this solution:

1. **Low-entropy for password based derived keys.** If the symmetric key used is dervided from a user memorizable password, then the entropy of the key is much lower, and therefore a hacker or even the a powerful cloud storage service could decrypt the data by brute-force guessing the password.

2. **Fault-tolerance for lost keys.** If the symmetric key is not password-derviced, it must be random and too long to memorize, hence the user must store it somewhere. This defeats the purpose of performing the cloud backup in the first place, as now if they key is lost due to a computer crashing, the data is irrecoverable.

3. **Usability.** Remembering passwords or storing keys is an extra hassle for the user, which makes system less usable. In fact, most users do not encrypt their personal cloud backup either because the software is too complicated or the extra burden is simply not worth the promised protection.

2 Introduction

With these motivating flaws of existing solution in mind, we present the design and implementation of Chasm, a secure cloud backup solution that is truly fault-tolerant and provides information-theoretic confidentiality and integrity based on Shamir's Secret Sharing Scheme.

The Chasm system is designed for the everyday user. Chasm has no passwords and provides a user-interface that everyone already knows: simply drag and drop a file into the Chasm folder, and the rest is taken care of.

Chasm operates over already existing, independently operated cloud storage services like Dropbox, Google Drive, iCloud Drive, Microsoft One Drive, or AWS, which most users today already have accounts with.

The security strength of Chasm is determined by N, the number of cloud services the user has delegated Chasm to use, and K the user chosen recoverability threshold. Chasm secret share the user's private data between N cloud storage services, setting the recoverability threshold to K using Shamir's scheme.

3 Threat-Model

Before we describe the details of Chasm, we first present our adversarial and threat models. We consider the following adversaries:

1. **A Malicious Cloud Storage Service** is motivated to read user data for a variety of reasons like system performance upgrades, market research, or possibly even to sell for more profits.

2. **A Hacker who breaks into a storage service** motivated to find valuable user data to expose for blackmail, sell, or even use to access bank accounts.

3. **A Nation-state actor** could be motivated to learn more about a political dissident or state enemy, spy on its own citizens, or spy on citizens of foreign countries.

Given these adversaries, we consider the following threats:

- Cloud storage services can directly read and copy any data that they are storing for the user

- Large cloud hosting services, nation state actors, or even hackers can have large computational resources, which may allow them to brute-force the passwords or weak keys

- A nation-state actor (like the NSA for example) can by legal means coerce K of the storage services to leak user-data.

- A nation-state actor may even by legal means require the cloud storage services to deny the user access to their data, thereby causing a denial-of-service attack.

Chasm is designed to defend against these adversial threats with the assumption that at-least K-out-of-N cloud storage services are not corrupted in these ways.

The fear of nation-state attackers is especially prevalent in Europe, following the revelations of Edward Snowden. There, concerns grow over the fact that most cloud storage services are US-based and hence susceptible to strong-arming by the American government. Many businesses hence seek alternatives or secure backup schemes in which the cloud storage service cannot necessarily be trusted [2].

4 Related Work

As mentioned previously, existing commercial backup schemes suffer problems of over-reliance on a password. But in addition to this, their numerous features add a layer of confusion between the user and the backup provider, and layers of complication magnify potential for human error to introduce vulnerabilities. In a sense, these services seem to offer a security-through-obscurity scheme in which the user simply has to trust that the services provide the promised security.

And even if they do live up to the promised standards, there continues to exist a vulnerability under our current threat model; namely, that exactly one organization controls the data. From a risk management point of view, increasing the number of organizations responsible and decentralizing via a secret sharing scheme can reduce the probability of a breach resulting in harm considerably, and increase the survivability of storage [12].

There exist a number of proposals in the literature for backup schemes of various styles. Each of these trades off different levels of confidentiality, availability, and integrity of data, especially with considerations for the performance of secret sharing schemes and the space costs of replication schemes.

One common technique is to increase availability in the case of a breach is state machine replication, whereby many machines provide the same service, in this case, backup the same data, and if during restoration, a majority of the machines agree on an answer, that answer is treated as correct [10]. In order to preserve confidentiality, the data needs to be encrypted before upload. However, in addition to being very costly space-wise, the need to encrypt data presents a number of problems. In particular, in a state machine replication scheme, key management is difficult to deal since all the data need to be redistributed [9].

Another, more efficient form of replication is a quorum system. Rather than have all the servers duplicate the data, in a quorum system each of the servers have some subset of the data such that they overlap in data with other systems. More specifically, each server has some nonempty intersection with every other server [8]. One such scheme that combines a quorum system with secret sharing distributes multiple shares to each server, such that they overlap and multiple servers have copies of the same shares, but no $x-1$ of the servers together have all of the shares and any x of them do [5].

While these replication systems are used to improve Byzantine fault tolerance, against situations even where servers arbitrarily deviate from what they are intended to do, they are bulky and somewhat inflexible. For example, share renewal in long-term storage systems is A number of solutions have been proposed that combine various replication systems with secret sharing [11, 6]. The motivation for such solutions primarily stems from the fact that secret sharing is a fairly computationally expensive process, as demonstrated by benchmarks performed by Subbiah and Blough (2005) [11]. To improve on various problems with pure secret sharing schemes, other improvements have been suggested. For example, POTSHARDS uses two layers of secret sharing to improve computational performance [3], while another scheme, which is more computationally expensive, allows verification of shares after they have been computed, and provides for simpler methods of share renewal [1].

There also exist schemes, which instead of secret sharing the data itself, encrypt the data with an AES key and secret share the key [4]. While such a scheme is much more computationally feasible, Subbiah and Blough (2005) points out that such a scheme still is susceptible to the weaknesses of the cryptographic keys itself, so compromise of the keys by brute-force or by holes in application security can reveal all the information [11].

Finally, while all these schemes exist within the minds and writings of academics, one example of secret sharing-based backup being used in enterprise systems is at the Kyoto University Hospital, where the confidentiality, integrity, and availability of patient records are all crucial to maintain, and the data must be recovered quickly in the case of database disaster [7].

Figure 1: Basic structure of Chasm. When a file is added to the Chasm folder, its contents are secret shared across all connected storage services.

5 How Chasm Works

The primary interface to Chasm is the command-line utility, written in Go, which can perform all of the following tasks. For specific usage information, run `chasm --help`. There is also a GUI under development.

5.1 Setup

The main step in setting up Chasm is connecting it to several cloud storage services, such as Dropbox or Google Drive. Chasm requires at least two to function and does support multiple accounts on the same service. The user is responsible for authenticating themselves to those services; in most cases, the service will provide a one-time code that the user can copy-paste into Chasm to allow it to read and write the user's cloud storage.

When Chasm is started for the first time, it creates a "Chasm folder" at the specified location (by default, the user's home directory) and initializes it with a `.chasm` file. The `.chasm` file stores all the information necessary for Chasm to function: authentication tokens for each cloud service and metadata about each file (see below for specifics).

5.2 Sharing/backup

The Chasm folder can be used like any regular directory: users can add, modify, and remove both files and directories. Chasm listens for all filesystem events within this folder; when a file is created or modified, Chasm splits it into n shares using Shamir's secret sharing scheme, and uploads one share to each cloud store. Removing a file causes all of its shares to be deleted from the cloud stores. Performance is not an issue: the Go implementation of Shamir's sharing can process multi-megabyte files for small n and k in less than a second; moreover, the sharing and uploading all happens in the background.

To reduce the amount of information available to an adversary who has access to a cloud store, Chasm assigns random IDs to each file and flattens the directory structure. These features are both implemented through the `.chasm` file, which contains a mapping between the IDs and the full original file paths. The only file not obscured in this way is the `.chasm` file itself, since Chasm needs to be able to find it first to restore the rest. One additional benefit of this design is that moving or renaming a file in the Chasm folder only requires resharing the `.chasm`, and not the affected file, since its content did not change.

5.3 Restoring

To restore the backed-up files, Chasm must first be connected to all the cloud stores as described in section 5.1. After that, Chasm can completely restore the file contents and directory hierarchy of the original Chasm folder. It downloads the shares stored on every cloud store, and restores `.chasm` first. It then recreates all the other files, and recreates the necessary folder structure according to the metadata in `.chasm`.

Shamir's secret sharing is a malleable scheme, so an adversary with write access to the cloud stores could corrupt the user's data. To prevent this, the `.chasm` file also includes the SHA-256 hash of each file, allowing Chasm to detect modified files. If $k < n$, then Chasm can also try different combinations of shares to figure out which share was corrupted and restore the file uncorrupted.

6 System Guarantees

Chasm is based off Shamir's Secret Sharing Scheme, which provides information-theoretic security. Therefore, we may make the following guarantees:

- **Confidentiality** As long as less than k out of n services collude, no adversaries can gain any information about the user's data. Therefore, we have information-theoretic confidentiality.

- **Fault-Tolerance** As long as the user still has access to data from k out of n services, she can recover any lost data when, for example, her computer crashes.
- **Integrity** The integrity of shares is verified by computing the SHA-256 hash of each share. Therefore, as long as k out of n shares have not been corrupted, the original data can be reconstructed.

The user can set the values of k and n as needed, determining the extent to which these three properties are upheld.

7 Usability

Chasm provides a simple solution for distributed secret sharing of potentially sensitive data. Because many users already have services like Dropbox and Google Drive, users can install Chasm with minimal overhead.

Similarly, the recovery process is extremely simple. For example, if a user's computer crashes, he can simple install Chasm on his new computer, log into his file storage services, and recover his data with a simple command. Restoration can happen at any time on any device. The only requirement is that users know the passwords to these services, which is a reasonable expectation. The user does not need to remember or store any encryption keys.

Finally, Chasm offers a simple and usable interface. Upon starting the Chasm process, users can simply drag-and-drop files into the Chasm folder to sync it across connecting file stores. A GUI for a desktop application is currently being created, which removes the technical overhead in using the command line and increases the simplicity in syncing, restoring, and logging into the different services.

8 Issues

There exist several known issues and security vulnerabilities, but fortunately their solutions are within reach.

In the current framework, it is very easy for an adversary to determine the number of files and size of each of the files. While the file structure and the names of each of the files is hidden, this information leakage could potentially be unacceptable to a highly paranoid user. The solution to this is to concatenate data and then redivide into blocks of fixed size, so that only the quantity of data being backed up is visible.

While the cloud storage services gain no information about the data since they only receive one of the shares, a potentially malicious service provider or anyone listening on the network on which backups are taking place might be able to look at the outbound shares. Fortunately, most of the cloud storage services that people use often use secure communication protocols to transmit data over networks.

While using secret sharing precludes the need for passwords or keys to encrypt data, Chasm still relies on users having secure accounts on cloud storage websites. The insinuation is that users might more easily remember their account information for commonly used websites such as Google, compared with their passwords for seldom-used backups that don't often require password inputs. This presents the potential problem that for users with poor password security on their various cloud storage accounts still have vulnerable; namely, adversaries can exploit weak or reused passwords to gain access to backed-up data if they are aware of the scheme that is being used. To improve security on this front is outside the realm of what Chasm can do for users; Chasm's security presupposes its users having adequate security on their cloud storage accounts. Luckily, for many users this is a simple fix - they can adopt longer, more unique passwords, and use two-factor authentication.

Researchers in backup systems often point to the fact that secret sharing is extraordinarily computationally expensive. Benchmarks done by, for one, Subbiah and Blough (2005) [11], show that for large values of n or x, or for very large amounts of data, processes such as polynomial interpolation in finite fields can grow very expensive. In that respect, Chasm is not designed necessarily for very large networks on servers used to communicate data; notwithstanding the needs of extraordinarily paranoid users, the use of up to $n = 5$ is far more than enough to provide the security needed under our threat model. Furthermore, Chasm is not designed necessarily for backup of large files like videos, and even if it is used for this, uploading is done in the background and will not be an extreme inconvenience to users. The restore function, since it is not done very often, is not bad if it is too costly anyway.

There exists, still, the possibility that an adversary, or a malicious cloud storage organization, might silently modify the shares. By sharing the hashes of the shares, Chasm is still able to detect these modifications, and it is computationally difficult for an adversary to modify data completely silently, assuming target collision resistance. If more than $n - x$ of the servers are compromised and data is modified, then Chasm can do nothing to recover the data, but the extent of this intolerance is no worse than Chasm's Byzantine fault tolerance or crash tolerance: if more than $n - x$ servers simply crashed, or did not return the right data, then the data is irrecoverable anyway.

9 Next Steps

Possible future additions to the core project concept include:

- Support for more storage services, allowing larger values for x and n
- Storing the uploaded files as blocks of uniform size to obscure their size
- Encrypting the data before uploading
- Detecting if the storage services have colluded (possibly not feasible)
- Increasing performance by parallelizing the generation of shares and uploading to the file stores
- Improving performance using methods outlined in any of [11, 9, 3]
- GUI for setup and login instead of command-line utility

10 Conclusion

In this paper we presented Chasm: an application of Shamir's Secret Sharing scheme that uses indepdent, pre-existing cloud storage services, like Google Drive and Dropbox, to provide everyday users with information-theoretic confidentiality, integrity, and fault-tolerance for sensitive data and file backup in the cloud. Chasm beats competing secure backup services in usability, security, and even availability while asking for almost zero trust beyond the correctness of the client side application.

Chasm operates in a more realistic threat model for today's ever increasing dependence on cloud services. Chasm distributes trust instead of relying on any single point of failure, a principle which we hope more services and application will start to adopt in the future.

References

[1] Chor, Benny, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. "Verifiable Secret Sharing and Achieving Simultaneity in the Presence of Faults." *26th Annual Symposium on Foundations of Computer Science* (1985): 383-95.

[2] Gastermann, Bernd, Markus Stopper, Anja Kossik, and Branko Katalinic. "Secure Implementation of an On-premises Cloud Storage Service for Small and Medium-sized Enterprises." *Procedia Engineering* 100 (2015): 574-83.

[3] Greenan, K., M. Storer, E.l. Miller, and C. Maltzahn. "POTSHARDS : Storing Data for the Long-term Without Encryption." *Third IEEE International Security in Storage Workshop* (2005).

[4] Herlihy, Maurice P., and J. D. Tygar. "How to Make Replicated Data Secure." *Advances in Cryptology - CRYPTO '87 Lecture Notes in Computer Science* (1988): 379-91.

[5] Ito, Mitsuru, Akira Saito, and Takao Nishizeki. "Secret Sharing Scheme Realizing General Access Structure." *Electronics and Communications in*

Japan (Part III: Fundamental Electronic Science) 72.9 (1989): 56-64.

[6] Lakshmanan, S., M. Ahamad, and H. Venkateswaran. "Responsive Security for Stored Data." 23rd International Conference on Distributed Computing Systems, 2003. Proceedings.

[7] Kuroda, Tomohiro, Eizen Kimura, Yasushi Matsumura, Yoshinori Yamashita, Haruhiko Hiramatsu, Naoto Kume, and Atsushi Sato. "Applying Secret Sharing for HIS Backup Exchange." *35th Annual International Conference of the IEEE Engineering in Medicine and Biology Society* (2013): 171-74.

[8] Naor, Moni, and Avishai Wool. "Access Control and Signatures via Quorum Secret Sharing." *Proceedings of the 3rd ACM Conference on Computer and Communications Security* 9.9 (1996): 909-22.

[9] Reiter, Michael K., and Kenneth P. Birman. "How to Securely Replicate Services." *ACM Transactions on Programming Languages and Systems* 16.3 (1994): 986-1009.

[10] Schneider, Fred B. "Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial." *CSUR ACM Computing Surveys* 22.4 (1990): 299-319.

[11] Subbiah, Arun, and Douglas M. Blough. "An Approach for Fault Tolerant and Secure Data Storage in Collaborative Work Environments." *Proceedings of the 2005 ACM Workshop on Storage Security and Survivability* (2005): 84-93.

[12] Wylie, Jay J., Michael W. Bigrigg, John D. Strunk, Gregory D. Ganger, Han Kiliccote, and Pradeep K. Khosla. "Survivable Information Storage Systems." *Computer* 33.8 (2000): 61-68.