Daruma: Regaining Trust in Cloud Storage

Doron Shapiro
CIS 2016

Michelle Socher
CIS 2016

Ray Lei
NETS 2016

Sudarshan Muralidhar
NETS 2016

Boon Thau Loo
Advisor

Nadia Heninger
Advisor

ABSTRACT

Currently, cloud storage services are used by consumers for a wide variety of important documents, including family photos, healthcare information and proprietary corporate data. These services all make promises about their storage solutions, usually including some guarantees of confidentiality, integrity, and availability. However, downtime is a fact of life for cloud services and, for better or worse, many providers openly admit to being able to access customer files for purposes ranging from analytics to law enforcement.

Daruma solves this problem by eliminating the need to trust any cloud provider. We run no servers ourselves - instead, we combine and secure the space on cloud services already used by consumers (like Dropbox and Google) with advanced cryptographic and redundancy algorithms. Our system provides a simple guarantee: no one cloud service provider can read, change, or delete your files – ever. Daruma feels just like an existing service - there are no extra passwords to remember or frustrating workflows to navigate. Daruma handles the complexities of security and reliability for users, allowing them to confidently utilize cloud storage without worrying about their previously inherent risks. ■

1. INTRODUCTION

Today, millions of people rely on cloud services to store their files. Companies like Dropbox Inc., Box Inc., and Alphabet Inc. have popularized the model of offering free or cheap storage capacity on servers they administer. These companies offer a convenient offsite backup and file-sharing service for their users and these users often take advantage of this by storing a variety of confidential or otherwise valuable documents on them. However, cloud storage necessarily come with a large set of risks. The ”cloud” is susceptible to a wide variety of failures, ranging from a user's spotty connection to the Internet to sophisticated hacking attempts and government interventions. Non-technical end-users rarely consider - or even know how to consider - what might happen were these cloud services to fail.

There are several naive approaches to guard against potential cloud service failures. One option is to encrypt all files before upload. However, such a process forces users to remember an encryption key; if this key is forgotten, all files are permanently lost. We wanted to build a solution that securely stored files and any encryption keys on the cloud, without leaking any sensitive data.

A potential solution to achieve fault-tolerance is to backup files on multiple providers. However, this is very space inefficient, as it stores the complete file on every provider used. We strove to build a system that combined cloud providers in a much more space-efficient manner, while still providing the availability guarantees of a backup protocol.

Tahoe-LAFS is a system implemented by Zooko Wilcox-O'Hearn that also attempts solve these problems by combining cloud providers [11]. However, Tahoe is targeted at system administrators and network security specialists. It is virtually impossible for an average user to set up, and requires users to be able to run code on their cloud storage providers. Thus, it is not a tool that can be used out of the box - it requires a significant time and money investment before its benefits become apparent.

We designed a system that accomplishes all of these goals in a verifiable manner while abstracting its inherent complexity behind a familiar interface.

2. APPROACH

2.1 Threat Model

One of our first steps was carefully developing a threat model for our application. We started by characterizing our users and split them into two groups based on need:

Business users Likely to be non-technical, but have proprietary corporate information that requires confidentiality and fast availability.

Personal users Similarly non-technical but might be motivated more by a sense of general privacy in addition to concerns of specific data sensitivity (e.g. tax form storage).

We then developed a list of adversaries and their capabilities:

Rogue Provider Providers are all corporations that administer servers to provide remote data storage. They have the ability to read, modify, or delete all files that a user stores, in addition to the ability to entirely remove a user's account, provide a third party with access, and collaborate with other providers. They may do this for a variety of reasons: they may be hacked by a third party, house malicious employees, be compelled by a government, see opportunities for business value, or make engineering mistakes that lead to these outcomes.

State-Level Actor Governments are increasingly expanding their electronic surveillance capabilities through technical and legal means. Today, it is widely assumed that a state-level actor can read, block, and modify network traffic. They can also in many cases legally compel corporations to perform many of the actions listed under *Rogue Provider*. These actions may be taken on both a large

scale in a dragnet-style surveillance effort or in a targeted manner for an individual investigation.

With this in mind, our primary goal was to protect the confidentiality, integrity, and accessibility of user file contents on computers other than the users' own. As an additional goal, we wanted to be able to apply these guarantees to file metadata where possible.

To do this, we set the following security guidelines:

(1) All network-bound traffic must be encrypted with a key providers do not know (i.e. SSL is insufficient for this purpose).
(2) All encryption must be authenticated to prevent tampering.
(3) It should be difficult for any one non-user actor to hold all the information necessary to reconstruct a key.
(4) It should be difficult for any one non-user actor to, by corrupting or removing access to data, remove access to a user file.

2.2 User Experience

One of our overarching goals was to make our application usable for non-technical end-users, so we had to develop an experience that made our features clear without forcing users to understand any of the behind-the-scenes implementation. We identified four key areas to focus on:

Key Management This project needed to keep track of many keys for both file encryption and provider authentication. However, users rarely make proper key selection or storage choices, so we decided that users should not have to remember any keys beyond their existing provider login credentials. Making all key management happen behind-the-scenes would also decrease the friction in adopting and using our system.

File Management Users needed a way to add, remove, edit, and read files tracked by the system. The filesystem interface handles all of these operations in a way that computer users are already very familiar with, so we decided we would either present a filesystem-like interface or operate on top of the existing filesystem. While the current implementation only supports the latter due to its ease of use and interoperability, the former may be more suited for certain archival purposes, as it allows an end-user to store files in the cloud that they may not want to store locally (perhaps due to local capacity constraints).

To provide the least friction in usage, we decided to implement the user interface pattern popularized by Dropbox of synchronizing a target directory in a user's filesystem and visually indicating the synchronization status of each file in the operating system file explorer (e.g. Finder on OSX).

Fault Handling This project is intended to be exposed to Byzantine faults across a distributed network, ranging from network failures to state-level attackers. Some of this behavior can be recovered from, while other behavior can result in more catastrophic failure. Moreover, the degree to which a system can recover from data is often configurable (e.g. by increasing redundancy). Since our target users were non-technical, we wanted our fault handling process to be as automatic as possible while still communicating to the user the status and risks of our system. As a motivating example, we considered what would happen if a file was corrupted by a provider. Our system could have been designed to be resilient this sort of error by just reconstructing the relevant user file with data from other providers and then notifying the user. However, there would still be many questions and decisions for the user at this point in time. Was the overall system still safe to use? Should they remove the offending provider? What should they do with the recovered file? To that end, we decided to implement a fault recovery algorithm that would go beyond the basics of merely detecting errors and recovering files to also securely fix errors as they came up. To help users understand the overall system health as we did this, we decided to implement a scoring system that would educate users about the behavior of their providers and advise them on how each one was affecting the system. Finally, we decided to make all of our operations atomic with automatic rollback to reduce the number of states our system could be in.

Performance To maintain ease of use, we wanted our users of our system to experience the same upload and download speeds that they maight see on their existing providers. Similarly, since we expected that many users would have free-tier provider accounts (with relatively low storage capacity), space efficiency was a high priority. We captured both of these constraints setting as a goal a network cost that increased roughly linearly with plaintext size.

2.3 Architectural Overview

Our architecture can be divided into several modules:

Manifest Module This module presents a filesystem abstraction to the user. Daruma-tracked files have numerous metadata points associated with them and are not stored internally in the same hierarchy that they are on a user's filesystem, so this module translates between our internal representations and a filesystem representation.

Encryption Module This module encrypts and decrypts all user file data, including the manifest. This allows us to guarantee confidentiality of all file data as well as a significant amount of metadata contained in the manifest, such as user file names. By using authenticated encryption, this module allows us to verify the integrity of data upon retrieval. All files are encrypted with different keys, all of which are stored in the manifest.

Distribution Module This module takes encrypted files, splits them into shares using an erasure encoding scheme, and disperses the shares among providers. Under this scheme, a configurable number of shares may be lost while still guaranteeing retrievability of data.

Master Key Module This module administers the randomly generated master key, which is used to encrypt the manifest. The master key is also split into shares using a variant of the Shamir Secret Sharing algorithm that provides both resilience to share deletion and share secrecy. When necessary, the master key is regenerated to protect secrecy.

Provider Module This module provides a common abstraction for all provider APIs.

Resilience Module This module runs on top of all of the other modules and handles recovery from provider errors. If a provider goes down or corrupts files, it can execute the relevant recovery by, for instance, redistributing files among the remaining providers or reuploading uncorrupted data to the faulty provider.

2.4 Filesystem Abstraction

To implement our system, we began by designing a secure filesystem representation that we would use internally to track user files. We needed the following three properties out of our representation:

(1) The filesystem should support storing files, files in directories, and empty directories. While there are other features that filesystems support (e.g. hard links), we found this to be a minimal set of features that gave us compatibility with average daily use.

(2) The filesystem should hide file metadata (e.g. directory structure and node names).
(3) The filesystem metadata should have a minimal storage overhead.

We decided on the following structure:

(1) All filesytem metadata would be stored in a single manifest file. This file would map user file paths (e.g. "/Users/alice/documents/foo.txt") to an internal codename (e.g. "5DB62955FBCD4228968A046A873A9236") as well as other metadata, such as encryption keys.
(2) User files would be stored in a flat structure in a single directory. Each user file would be stored under its code name rather than its file path, thus hiding file paths from providers.
(3) To achieve atomicity, we would treat user files as being immutable and change the codename references in the manifest after we had confirmed that any given operation had succeeded. For more details on this implementation, see appendix B.1.

2.5 Authenticated Encryption

We chose Dan Bernstein's NaCl package for authenticated encryption [3]. We selected this package for several reasons:

Authentication This package offers authenticated encryption, which was pivotal for our ability to detect and punish provider misbehavior. This guarantees the integrity of our system because successful decryption of authenticated ciphertext ensures that the data has not been modified.

Vetted Cryptography Dan Bernstein's work is highly respected within the cryptography community and this package has been vetted by those in that community.

No User Modifications The goal of this package was to provide usable cryptography where all details and decisions are removed from users of the library. It is generally considered poor practice to roll one's own cryptography or even attempt to navigate the various settings and parameters without strong expertise.

Using this package, we generate a random key for each file. The mapping from files to keys is then stored in the manifest. It was important that each file get its own key (rather than a single master key being used to encrypt all files). Otherwise, our scheme would reduce to e.g. AES-ECB, an AES mode that is known to reveal metadata about the data it encrypts because all identical blocks encrypt to the same ciphertext.

2.6 Erasure Encoding

One of our main goals was to efficiently combine the space available to users through a single and secure interface. We therefore needed to distribute user files in such a way that we could maintain confidentiality, reliability, and availability. This corresponds to our core promise to users: no provider can read, delete, or modify their files.

We therefore needed an overall distribution scheme that would guarantee these properties while being both time and space efficient. In our own research, we found the following primitives and schemes available for such use:

All-or-Nothing Transform (AONT) This is an *s*-of-*s* threshold scheme; all shares are required for the reconstruction of the original secret. As a result, the algorithm runs quickly, in $O(\log(s))$ time for a message of length s, and no additional storage costs are incurred. This is often used as a preprocessing step for separable encryption suites such as block ciphers (e.g. AES-CBC) where decryption of a single block of ciphertext results in a single block of plaintext. This scheme ensures that no information is revealed unless all blocks can be decrypted, protecting against certain brute force attacks. [9]

Rabins Information Dispersal Algorithm (IDA) This dispersal algorithm guarantees that any *m* of *m* shares can be used to reconstruct the secret. It runs in $O(m^2)$ time and the total space required for dispersing data of size F is $\frac{n}{m} * F$. If the parameters are set so that $n = m$ then the dispersal incurs no additional storage cost, consistent with AONT. Since each share will be $\frac{1}{m}$ of its original size, this dispersal achieves maximum space efficiency for its threshold. However, this also implies that the scheme does not make strong confidentiality guarantees - that is, some smaller than *m* subset of shares may reveal information about the secret. [6]

Shamir Secret Sharing Shamir Secret Sharing is very similar to Rabin's IDA but with some key differences. In particular, it is a *k*-of-*n* threshold scheme such that *k* share are required for reconstruction and any subset of fewer shares reveals no information. In order to make this guarantee, it is necessary that each share be the same size as the original secret. Therefore, for data of size F, the total stored data for this scheme would be $n * F$. [10]

Reed-Solomon Encoding This scheme makes the same guarantees as Rabin's IDA with the same space performance. Technically, Reed-Solomon encoding takes a message of length *m* and extends it to be *n* symbols such than any *m* symbols can be used for full reconstruction while the IDA actually generates shares. However, the IDA is discussed primarily in academia while there are a plethora of Reed-Solomon implementations (which follow the technical behavior of the IDA instead in terms of share generation). [5], [8]

Secret Sharing Made Short This scheme combines the confidentiality properties of Shamir Secret Sharing with the space efficiency of Rabin's IDA. Specifically, a random key is generated and used to encrypt the message that is meant to be kept confidential. The encrypted data is then distributed with Rabin's IDA and the key itself is protected and distributed with Shamir Secret Sharing. [4]

We were excited to find in our research that the Secret Sharing Made Short Scheme closely mirrored parts of the solution scheme we had independently been discussing. We chose this scheme because it matched the confidentiality guarantees that we needed while still providing optimal space efficiency for user files.

We had to make a few modifications on top of this scheme in order to match our use case. In particular, we implemented a stronger variant of Shamir Secret Sharing (see 2.7) and we used an existing Reed Solomon Encoding library. Additionally, rather than generating a single key in order to encrypt a single file, we had to generate a key for each file and store the mapping in a manifest (see 2.5). The manifest was then encrypted with a randomly generated master key. We then used Reed Solomon encoding to distribute the encrypted files as well as the encrypted manifest and distribute the master key with Shamir Secret Sharing.

2.6.1 Dependency Sandboxing. We used the PyECLib library [2], a wrapper around the liberasurecode library [1], to provide our erasure encoding capabilities. During the development of our project, we frequently found that by corrupting the shares provided to the library, we could induce a segfault in it and bring down our

entire program alongside it. To mitigate this, we reported the issues upstream to the PyECLib and liberasurecode maintainers, who helpfully developed fixes to the problems we identified. However, since the erasure decoding stage of our pipeline necessarily took raw data from providers as input, we were wary of undiscovered bugs in this library continuing to crash our program. So, we developed a sandbox for this and other library dependencies that would run their stages of our data pipeline in separate processes and report their results back to the main process via inter-process communication channels. With this development, any third-party library we depended on could crash (or be otherwise compromised) without harming our main application logic - any errors would be passed to the resilience subsection (see 2.9).

2.7 Robust Secret Sharing

In order to encrypt our manifest without forcing users to remember and protect a new credential, we needed to generate a random master key that could be stored across the providers with confidentiality, integrity, and reliability.

For confidentiality and reliability, we needed a threshold scheme such that for shares distributed across n providers, reconstruction of the secret would be possible with any k of these providers but no information about the secret would be revealed to any subset of fewer than k providers. These requirements were satisfied by Shamir Secret Sharing (see 2.6, [10]). However, Shamir Secret Sharing does not provide the integrity guarantees that we needed. In particular, Shamir Secret Sharing is tolerant to some subset of missing shares, but it is vulnerable to corrupt shares. We therefore needed to apply a verification wrapper around this scheme that would guarantee the integrity of the reconstructed secret.

Such schemes require metadata to be transmitted as part of the shares. Much of the academic work within this space has been targeted at minimizing the size of the resultant shares as the number of players increased. However, since we only needed to apply this scheme to a single encryption key and our target use case would have no more than 10 providers, we favored simplicity over asymptotic efficiency. We therefore chose the "Verifiable Secret Sharing" scheme presented by Rabin and Ben-Or [7].

For additional information on the inner workings of Shamir Secret Sharing and the Robust Secret Sharing wrapper, see appendix A

Open-source implementations of Shamir Secret Sharing are available and we had originally planned to make use of an existing library. We selected a library based on its apparent code quality and hygiene, but when we implemented our own unit tests we found a security flaw within that library. After communicating with the team behind that library, we decided that our best option would be to implement Shamir Secret Sharing ourselves. Furthermore, we were unable to find any Robust Secret Sharing implementations as this space has been largely academic. As best as we know, ours is the first Robust Secret Sharing implementation in the wild.

2.8 Providers

Daruma currently supports four popular customer facing cloud storage companies: Dropbox, Google Drive, OneDrive and Box.

We created two flows for creating providers. The first supports providers that implement the OAuth flow, and redirects the user to a link where they can log in on the providers' website. The second supports providers that take a single parameter (for instance, a provider residing on a local disk might require only the providers' path on disk for construction).

After providers are created, all providers share a common interface, so that the internal system can treat them all same way. All providers support GET, PUT, DELETE, and WIPE operations. These functions each wrap calls to the provider API.

OAuth tokens and credentials for providers are cached in a user credentials JSON file on the user's disk. This allows Daruma to automatically load cached providers on load, simplifying the user experience.

2.9 Resilience

A major concern when implementing Daruma was ensuring that no coalition of providers, through action or inaction, could corrupt the state of the filesystem. This included cases where certain providers strategically go offline during crucial uploads - for instance, if only some providers are online during an upload, it stands to reason that different parts of the system could be out of sync. For this reason, it was necessary to make all write operations atomic. Similarly, reprovisioning (redistributing files and shares upon changing a threshold or adding/removing a provider) was written such the system did not become corrupted if a provider failed during the operation.

In order to ensure that our guarantees were maintained as various cloud providers failed and came back online, it was necessary to quickly detect and repair errors on providers as soon as possible. Daruma needed to account for the fact that a provider could corrupt files at one time frame and thereafter behave correctly as some other provider failed. Both our major recovery/distribution protocols - Robust Secret Sharing and Reed-Solomon, for master keys and files, respectively - were written to provide, upon successful recovery, information about which recovered shares were invalid. The providers with invalid shares (and providers who did not return a share at all) were tagged as failing, and upon recovery, repaired. The repair itself consisted of re-encrypting (with a new key) and redistributing a file (in the case of an invalid Reed-Solomon share) or creating and sharing a new master key, and then redistributing the manifest (in the case of an invalid Robust-Secret-Sharing share. It is important to note that, in order to ensure that information was not leaked, new keys had to be used on all repairs; if not, a malicious provider could pretend to lose shares in order to collect multiple shares of the same plaintext.

In the case of a permanently failing provider, it was important that we have a limit to retries, after which we would decide not to continue repair attempts. It was also crucial that we report to users when providers were failing badly, so that such a provider could be removed. However, we needed a way to differentiate between providers who were experiencing temporary difficulties (and failed several times in a short time span, but resumed normal service afterwards) and providers who exhibited patterns of failure over time. To do this, we used an exponential smoothing formula of the form $s = \alpha x + (1 - \alpha)s$ for a provider score. Here, s is the provider's *score*, and x is a data point - 1 if a provider responded to the most recent request without errors, 0 otherwise. This score represents the amount of time a provider is responding without flaws, and accounts for both past and current behavior, weighting the latter more heavily. Below a certain threshold, a provider is considered "red" - retries are no longer attempted, and the user is advised to remove the provider. Below a higher threshold, a provider is considered "yellow" - the provider has been experiencing failures, but seems to be mostly okay. Tweaking the thresholds and α enable us to account for a wide variety of provider behaviors - either penalizing harshly or being more tolerant, as necessary.

For more information on the resilience protocols and how exponential smoothing parameters were chosen, see B.

3. RESULTS AND MEASUREMENT

3.1 User Interface

Our goal was to provide easy integration with a user's existing filesystem. Below is a comparison of our interface with the traditional Dropbox interface:

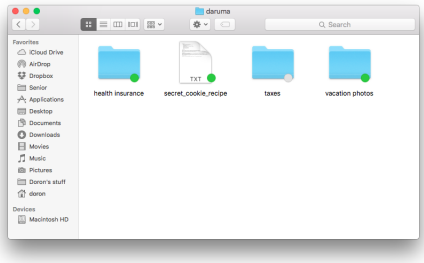


Fig. 1. Daruma Finder Integration on OSX

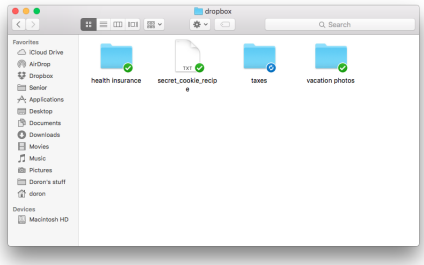


Fig. 2. Dropbox Finder Integration on OSX

3.2 Capacity Utilization

We broke capacity down into three possible categories:

Available Secured This represents the space that will be available to users with a redundancy guarantee.

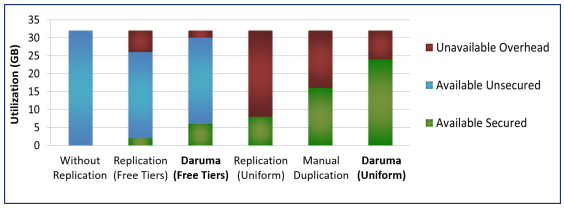
Available Unsecured This is space that is still available to users on their individual providers. They can access this space through the provider interfaces that they have always used but without any protection through redundancy.

Unavailable Overhead This space is lost and unavailable to the user for storage. This is the cost of redundancy for each scheme.

We also considered two different cases to show how Daruma's comparative performance varies across the capacity distribution of the providers:

Free Tier This shows the capacity utilization in the case where a user has accounts on the free tier of each of the providers. This gives them 2 GB from Dropbox, 5 GB from OneDrive, 10 GB from Box, and 15 GB from Google Drive.

Uniform This assumes a uniform capacity distribution across providers. In order to match the 32 GB total from the Free Tier category, we assumed here that each of the four providers would have 8 GB.



We showed the breakdown of the total capacity usage in four different cases (in two of those cases, the breakdown did not vary from the Free Tier distribution to the Uniform distribution while in the other two it did and both versions are therefore shown). We considered the following four distribution schemes:

Without Replication Users store files separately on each of their providers without any redundancy. This represents the way users currently interact with cloud providers.

This will give the greatest possible amount of unsecured available space but no secured space.

Replication This is the case of naive full redundancy. Each provider will have a full copy of each user file.

This will always be the inverse of Daruma - that is, its overhead will always be equal to the secured space on Daruma and Daruma's overhead will always be equal to its secured space when they are applied to the same capacity distribution.

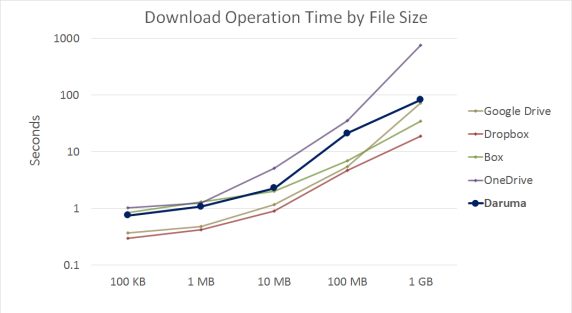
Manual Duplication Each user file is stored in full on two providers.

This provides more secured space than Daruma can on a Free Tier, but will significantly more overhead that consumes the large amount of available unsecured space that Daruma can still offer. It offers significantly less secured space and much more overhead as compared to Daruma used with Uniform capacity distribution.

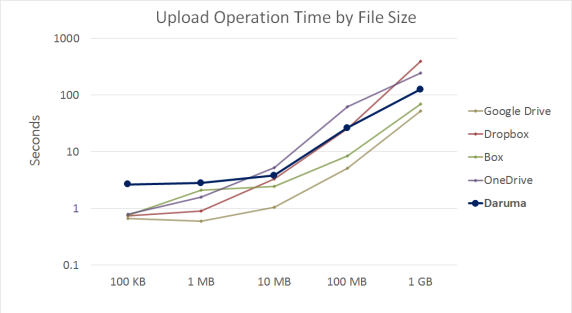
Daruma We use Reed Solomon Encoding to efficiently distribute user files under a variety of capacity distribution models. Our goal is to minimize overhead while securing as much space as possible.

3.3 Speed

Daruma's measured speed performance is quite good compared to using providers directly. All operations involve parallel network requests to all providers, so Daruma's speed is fundamentally affected by the speed of the slowest provider. However, because of the space efficiency of Reed-Solomon encoding, Daruma shares are smaller than the original files, resulting in some speed gains. In the following tests, Daruma was configured with 5 providers (4 online), and compared to file operations directly on providers. Note the the axes are in a logarithmic scale.



In a download operation, Daruma simply has to download Reed-Solomon shares from providers. The size advantage of using these shares becomes more apparent as file sizes grow larger, as Daruma becomes significantly faster than the slowest provider by the time file sizes approach 1GB.



In an upload operation, Daruma first uploads the file, and then uploads an updated manifest. For small files, the extra operation causes Daruma to be slower than other providers, but only by a few seconds (about 2 seconds slower for 100kb files). As files grow larger, the file size advantage described previously becomes more influential, and Daruma again becomes significantly faster than the slowest provider.

4. ETHICAL AND PRIVACY CONSIDERATIONS

4.1 Social Context

Online security and privacy have gained increased public scrutiny recently due in part to stories of large corporations being hacked and revelations of mass surveillance by governments around the world. While this is a rapidly evolving situation, several points remain clear. First, regardless of the risks, users and businesses are willingly trusting more and more of their data to the cloud. Secondly, the threat model a security-conscious company must maintain needs to include itself as an adversary, either due to the possibility of a rogue employee or because it may become the target of hackers or governments.

This latter point was recently highlighted in the high-profile legal battle between Apple, Inc. and the Federal Bureau of Investigation. During the case, it was revealed that Apple had implemented two versions of PIN protection in different generations of phones it manufactures and only in the earlier generation could it provide a tool to bypass the protection. While the case sparked a wide debate over whether the FBI should compel Apple to produce such a tool, the conclusion for Apple was clear: in the earlier generation, its position of trust made it an adversary to complete security.

4.2 Architectural Considerations

This conclusion was deeply considered in the architecture of Daruma. First, all Daruma code runs on users' computers. This means that if an adversary were to try to compromise Daruma centrally, they would no central surface to attack: there are no Daruma servers. There is still, however, the possibility that we as project developers can write or introduce malicious code in the project, either due to malintent, as the result of a legal order, or because of hacking. To protect against this, the entire Daruma codebase is published as an open-source project so that it can be audited before use. Even if an average end-user does not have the technical know-how to verify that our code operates as advertised, this opens up the opportunity for trusted third-parties to inspect the code and publish their results.

Finally, cloud security is a very complex landscape that often outstrips the technical understanding of its users. Because of this, we spend significant effort making sure that Daruma could provide all of its features to an entirely non-technical end-user. There are solutions that achieve some of the same goals that Daruma does with significantly more user maintenance and understanding, but we strove to ensure that traditional maintenance details, ranging from key management to error handling, were handled automatically.

5. DISCUSSION

Our current product is capable of replacing an application like the Dropbox client on a user's computer for most non-social tasks on a day-to-day basis (e.g. barring collaboration and link sharing). Users can currently log in with their credentials for Dropbox, Google Drive, OneDrive, or Box in addition to using standard filesystem paths as local providers (e.g. to use a mounted local backup drive). Once logged in, the application will watch a Daruma folder in the user's home directory for changes and synchronize the files it stores online with the Daruma folder state. When providers go down or otherwise remove access to or corrupt files, we properly recover the system if at all possible.

There are still some inherent weaknesses in the system as well as areas that we see opportunities for improvement in. We break these areas into the following categories:

Threat Model Weaknesses Our system has a threat model that is intentionally limited to only consider parties outside the user's computer as potential attackers. While this covered both providers and ourselves, we do not take significant steps to protect sensitive material on the computer system we are running on. If we were to expand our threat model, additional thought might be put into how to sandbox our application from local threats.

Usability Usability was a high priority for us as we developed, so significant effort was put into making interactions feel familiar for a non-technical user. However, there are other friction points that might be considered, such as our requirement that users have many existing cloud provider accounts. To mitigate these issues, future work could include making it easier to sign up for new provider accounts as well as more informative communication regarding system statistics such as capacity utilization.

Sharding When sharing files, Daruma currently builds shares for the entire file at once. For large files, this results in a significant memory cost. This can be avoided by cutting files into many small pieces (shards), and sharing these sharesd individually. The shards would then be reconstructed on download. If parallelized, a sharding operation would also significantly improve speed, as it would allow Daruma to upload multiple parts of a file at once.

Sharing While our system rivals Dropbox and other cloud providers in usability, Daruma lacks certain features that have become fundamental for other cloud providers. For instance, files sharing (allowing other users to view your files) is a feature commonly used on Dropbox, but unavailable in Daruma. Such a feature could be implemented if, on sharing a file with a secondary user, some public key and manifest information was passed to the secondary user. This feature would further help users transition from individual providers to Daruma.

Cross-Computer Usage Currently, our usage model assumes that users will not have concurrent Daruma sessions on two different computers. If this assumption was broken, we would have to implement several safeguards to ensure the systems do not go out of sync or enter a corrupted state if providers maliciously fail. While hard, these problems are not intractable, and would make Daruma more usable for everyday users.

Cross-Platform Interfaces Currently, Daruma's user interface only works on OSX. Making Daruma's GUI usable on all platform is a major future goal.

6. ACKNOWLEDGMENTS

We would like to thank our advisors, Nadia Heninger and Boon Thau Loo, as well as CIS senior design instructors Ani Nenkova and Jonathan Smith, for the invaluable technical advice and logistical direction they gave us throughout this process. We would also like to thank Brett Hemenway for his inspiration during our brainstorming process and for his in-depth technical guidance as we implemented the project. Finally, we extend our gratitude to Thuy Le for designing our logo and Melanie Wolff for helping us prepare our demonstrations.

7. REFERENCES

[1] liberasurecode. `https://bitbucket.org/tsg-/liberasurecode/`.

[2] PyECLib. `https://bitbucket.org/kmgreen2/pyeclib`.

[3] Dan Bernstein. NaCl: Networking and Cryptography library. `https://nacl.cr.yp.to/`.

[4] Hugo Krawczyk. Secret sharing made short. In *CRYPTO '93 Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology*, number ISBN:3-540-57766-1, pages 136–146, 1993.

[5] R.J. McEliece and D.V. Sarwate. On sharing secrets and reed solomon codes. In *Communications of ACM*, number 24-9, pages 583–584, September 1981.

[6] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. In *Journal of the ACM*, number 36-2, pages 335–348, April 1989.

[7] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, number ACM 0-89791-307-8/89/0005/0073, pages 73 – 85, May 1989.

[8] I. S. REED and G. SOLOMON. Polynomial codes over certain finite fields. In *Journal of the Society for Industrial and Applied Mathematics*, number 8-2, pages 300–304, June 1960.

[9] Ronald L. Rivest. All-or-nothing encryption and the package transform. In *Lecture Notes in Computer Science*, number 1267, pages 210–218, May 2006.

[10] Adi Shamir. How to share a secret. In *Communications of the ACM*, number 22-11, pages 612–613, November 1979.

[11] Zooko Wilcox-O'Hearn. Tahoe-LAFS. `https://www.tahoe-lafs.org`.

APPENDIX

A. IMPLEMENTED CRYPTOGRAPHY

For our project, we implemented two core cryptography schemes.

A.1 Shamir Secret Sharing

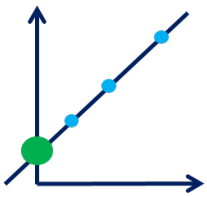
This is a *k*-of-*n* scheme such that any *k* shares can be used to reconstruct the secret but that any subset of fewer shares reveals no information. This is fundamentally achieved with polynomial evaluation and interpolation [10].

Sharing begins with the generation of a polynomial *P* with *k - 2* random coefficients and the y-intercept set as the secret to be shared. The polynomial will therefore be of degree *k - 1* and *n* x-values will be selected for evaluation. The resultant (x, *P*(x)) points are then distributed as the shares associated with the secret [10].

For reconstruction, a minimum of *k* of these points can be used to reconstruct the polynomial *P*. By taking *P*(0) it is straightforward from there to recover the secret [10].

We can demonstrate how this scheme works with motivating example. We consider a case with 3 players where any 2 of them can be used for reconstruction. Our polynomial with the secret as the y-intercept will therefore be of degree 1 (i.e., a line). We see from this diagram that with any two shares (points) we can reconstruct the line and therefore the secret. However, an infinite number of lines could pass through a single point so with one share no information about the secret is revealed.

Fig. 3. n=3, k=2



While this scheme protects well against missing shares, it is vulnerable to corrupt shares. In that case, polynomial interpolation will construct from the points a polynomial different from the one originally generated, preventing the recovery of the secret. This problem is addressed by Robust Secret Sharing.

A.2 Robust Secret Sharing

We used Robust Secret Sharing so that we could tolerate and identify up to *k - 1* corrupt shares. The algorithm proceeds as follows -

Shares are generated as before through Shamir Secret Sharing. Each of these shares is then used as the message for *n* generated check vectors. A check vector is therefore generated for each pair of
, and this metadata is sent to the providers along with the generated shares.

When the providers return their shares and metadata, we use that metadata to create lists of verified shares from the perspective of each provider. This allows us to use Shamir Secret Sharing for each such list to reconstruct what each provider would think the secret would be if they had access to the shares and metadata.

We then apply a voting scheme to select the correct secret.

If fewer than *k* providers returned corrupt shares and we have at least *k* honest shares, the secret returned from this process is guaranteed to be the one that was originally shared.

This scheme as described varies slightly from the algorithm presented by Rabin and Ben-Or [7]. In particular, their algorithm guarantees that upon reconstruction all players broadcast their information and the guarantee they make is that each honest player will correctly recover the secret originally shared [7]. However, we do not ever want the providers learning the original secret so rather than having players that broadcast their data to each other upon reconstruction, we request information back from each provider. We then take a vote on the secrets constructed from the view of each provider, and the correctness of this scheme reduces to the guarantees made in that paper.

B. RESILIENCE

B.1 Atomic Algorithms

In order to guarantee that sudden provider failures would not put the system in an unstable state, all algorithms needed to be atomic (all-or-nothing). For put and get operations, this was achieved by using a manifest update as a ”commit” operation.

Algorithm 1 Put *file*

(1) Share *file* under new random name
(2) Update manifest to point *file* to the random name
(3) If *file* existed previously, delete files with the old random name

Algorithm 2 Delete *file*

(1) Update manifest to remove *file*
(2) Delete files with the random name previously pointed to by *file*

In each of these operations, a failure before or after the manifest update results in a consistent state across all providers (with perhaps some garbage files that need to be deleted). While manifest failures do have the potential to bring the system out-of-sync, our threat model assumes that a maximum of *threshold* providers fail at a time. If this is the case, then the manifest operation is committed, and the manifest will be repaired on later operations as the failing providers become operational. This random-name scheme makes any operation reversible because there is a clear ”commit” step, without which all providers are left unfinalized.

B.2 Reprovisioning

As a result of our assumption that providers can go offline or out of business at any time, we needed to make it possible for users to remove providers from the system and replace them with new ones. However, it was crucial that we maintain all guarantees after such a replacement. To do this, all files and the master key needed to be reshared across providers with the new threshold parameters. Atomicity was achieved by using a manifest-commit operation similar to Put and Wipe. As before, failures in steps (1) and (2) result

Algorithm 3 Reprovision

(1) For every file in the system, recover it from the old provider set, and reshare it to the new provider set with a new name
(2) Create a new manifest with information about all new shares, and share it across new providers
(3) Share the new master key across all providers, and broadcast the new manifest name (commit).

only in creating some extra garbage, and a failure in step 3 results in a consistent state that can be repaired.

B.3 Exponential Decay

After a successful operation, all providers are collected so that their internal scores can be updated. The scores are updated with an observation $x_{p,t+1}$, representing the providers' performance in the last operation. If the provider was successful, $x_{p,t+1} = 1$, and if the provider failed (invalid share, connection failures, etc), $x_{p,t+1} = 0$. The score is then updated according to the update rule

$$score_{p,t+1} = \alpha score_{p,t} + (1 - \alpha)x_{p,t+1}$$

For our purposes, α was chosen to be 0.7. Thus, the score is a reflection of both the provider's past and current behavior. If a provider's score drops below a threshold of 0.05, it is considered RED, and users are alerted that, while the system is fully operational, there are major problems with the provider. For scores between 0.05 and 0.95, users are notified that the provider is experiencing difficulties. If a provider is red, repairs and retries are not attempted. However, any requested operation is attempted at least once in all cases, ensuring that if a failing provider comes back online, its scores will rise back up.

B.4 Repair and retry

If an operation fails, all identified failing providers are updated according to the exponential decay update rule. Then, if no providers are RED, the operation is retried. Because a failure necessarily decreases the score of at least one provider, the retry procedure will always halt - either with a successful operation, or with a provider becoming RED.
If an operation is started when some providers are RED, the operation is still tried once. Then, on failure, the operation is not retried, as retries only happen when all providers are not RED.
When a provider loses or corrupts a file/key share, that share needs to be replaced as soon as possible. Upon a successful recovery and diagnosis of failing providers, any necessary repairs are performed. For failed key shares, this involves choosing a new master key, re-encrypting the manifest, and re-sharing and re-distributing the bootstrap keys and manifest. For failed file shares, the repair involves re-uploading the file, encrypted with a new key and stored with a new random name. These repairs follow the same protocols as any other operations, and so ensure that the system stays in a stable state, regardless of any mid-operation provider failures.