Improving the Security of Static Data on Cloud Storage Platforms

Zachary Daniel Leaf
School of Electronic Engineering and Computer Science
Queen Mary University of London
London, UK
zdleaf@zinc.london

***Abstract*—There are significant security risks associated with storing static data on cloud storage platforms. This paper examines these risks, presenting practical solutions to mitigate these. In particular, the focus is on the application of sound cryptographic principles to ensure the confidentiality, authenticity and integrity of stored data. This will be of use to software developers or organisations who intend to use public cloud storage to store data.**

The shortcomings of existing solutions are also explored and improved upon.

The result of this paper is the development of an open source tool for securely syncing files between a Linux machine and cloud storage platforms, with a focus on ensuring the confidentiality, authenticity and integrity of data. The full code is available at https://github.com/zdleaf/enclone

***Index Terms*—Cloud computing; Data privacy; Encryption; Cryptography; Secure storage**

I. Introduction

As Infrastructure as a Service (IaaS) and cloud storage have now become common place, this presents significant security risks associated with storing data on servers outside of the user's control. The concern over data security and privacy has been said to be the biggest hurdle facing cloud adoption (Kaufman, 2009; Kamara and Lauter, 2010).

A main benefit of cloud storage is that it is abstracted from the physical implementation of disks and server administration. Users can conveniently and instantly access large amounts of storage space, with a great deal of flexibility and reliability, without having to be concerned with the underlying infrastructure, maintenance and admin costs (Kaufman, 2009; Yan et al., 2018). However, the result of this is that user does not know where their data is stored, or what processes are running on that server (Kandukuri et al., 2009). Users are now reliant on third parties to protect and secure their data against security breaches, from both internal "insider" and external attacks, and both hardware and software vulnerabilities that potentially expose confidential data. Worse still, the virtualisation software used by cloud providers exposes completely new and additional vulnerabilities on top of existing threats (Aiello et al., 2014).

It is also often unclear how the data stored on these services is used or shared by both the provider and their third-party affiliates. This is particularly true for personal cloud storage solutions such as Dropbox and Google Drive. For example, the Dropbox Terms of Service clearly and explicitly states it gives both Dropbox and their affiliates and "trusted" third parties full access to store and scan uploaded data (Dropbox, 2019). Further, the physical or operational resources may be themselves outsourced to third parties in a way that is not always transparent.

The overwhelming majority of solutions are, by default, insecure out of the box and vulnerable to a number of risks without additional hardening. It has therefore become essential to understand these risks and design and implement a strategy to ensure the confidentiality, authenticity and integrity of data stored on cloud storage platforms. A key point here is that trust is not synonymous with security. Any security that is solely reliant on social means, i.e. "legal, physical or access control mechanisms" (Kamara and Lauter, 2010) such as promises made in Terms of Services or relying on the trustworthiness of a provider's employees to not access confidential data is insufficient (Kaufman, 2009).

The purpose of this paper therefore, is to evaluate and implement technical and practical cryptographic methods to guarantee as far as possible the security of data stored on these services, while retaining the benefits of public cloud storage systems. The findings are implemented in a Linux utility/daemon to sync/backup files and directories from a local file system to public cloud platforms, in a cryptographically secure way.

II. Potential Risks

A. External attacks

The number of potential external attacks are large. Due to the reliance of cloud providers on virtualisation to maximise server utilisation, data is stored in shared environments. Attacks on the hypervisor, the software that handles these shared environments, such as Virtual Machine (VM) hopping and escape attacks (Aiello et al., 2014; Alani, 2017; Geffner, 2015) allow attackers to escape the VM into the host system or to other VMs on the same hardware. The VENOM vulnerability (CVE-2015-3456) is a recent and infamous example of how it is possible to "escape from the confines of an affected virtual machine (VM) guest and potentially obtain code-execution access to the host" (Geffner, 2015). In this way traditional security vulnerabilities can be used to exploit and gain access to poorly configured or vulnerable VM instances to gain access to other VMs on the system and the wider network.

Side channel attacks, such as exploiting the shared physical memory on systems has allowed attackers to extract private keys and other information from VMs running on the same hardware (Alani, 2017; Zhang et al., 2012). The most infamous of such side channel attacks are the MELTDOWN (Lipp et al., 2018) and SPECTRE (Kocher et al., 2020) vulnerabilities that allowed attackers to extract information from other processes running on the same CPU. This is a critical security issue in a multi-tenant cloud server containing confidential information.

Access to cloud platforms typically includes a web-based interface as well as an API which provides further vectors for attack (Yan et al., 2018). In almost all cases full access to stored files is possible through such web interfaces. Web based vulnerabilities are numerous, ranging from SQL injection, Cross Site Scripting (XSS), Cross Site Request Forgery (CSRF) and exploitation of session management, auth tokens and authentication systems. There is also, of course, the possibility for misconfiguration of cloud services by the user or compromise of user credentials through social engineering or phishing. All of these are additional vectors exposed by such web interfaces and APIs.

B. Internal "insider" attacks

In the Frequently Asked Questions (FAQs) for Amazon Web Services' (AWS) cloud storage platform - S3, it states that Amazon does not "access your data for any purpose outside of the Amazon S3 offering, except when required to do so by law" (AWS, 2020*b*). However there remains the possibility that any insider working for AWS with sufficient permissions can access your raw data as they have access to the physical memory.

Similarly, while Dropbox's marketing material states that data is fully encrypted, the encryption keys are stored by Dropbox, giving them full access to your data at any time. Dropbox clearly states that a number of employees have full access to user data (Dropbox, n.d.). This opens the possibility for a "malcious insider" attack – that someone working within the cloud storage provider is be able to access to that data (El Makkaoui et al., 2017; Kandias et al., 2007; Sun, 2018).

C. Network attacks

As data is transmitted across open computer networks to public clouds, there is the risk of Man in the Middle (MITM) and network sniffing attacks. SSL/TLS offers protection on data transmitted across most open networks, however the transmission on the internal networks of cloud storage providers is unknown. Further there have been previously been serious vulnerabilities in the implementation of SSL/TLS, most notably the Heartbleed vulnerability (CVE-2014-0160) (Synopsys Inc, n.d.). This allowed security researchers to extract secret keys, username and passwords from SSL/TLS traffic.

D. Deduplication

Harnik et al. describe both file level and block level deduplication that is implemented by cloud storage providers. By eliminating redundant data, this enables cloud providers to maximise utilisation of the available hardware and storage drives. Harnik et al. show that this opens up a side-channel, leaking information about both the existence of certain files, and the contents of files (2010).

E. Legal and regulatory requirements

Kamara and Lauter outline three areas in which storing static data on cloud platforms can have legal and regulatory implications (2010). If a company has to store certain personal information, such as medical or financial records, there are usually laws regarding the safe guarding of this data. Kamara and Lauter state that therefore "the use of a public cloud storage service can involve significant legal risks" as "organizations are often held responsible for the actions of their contractors". Similarly, with different jurisdictions and geographic restrictions on storage of data, it can also be impossible to know in which region your data resides on cloud platforms. Lastly, Kamara and Lauter explain how data on cloud platforms can be subject to subpoenas without notifying the user. As providers store the data of multiple users on the same disks, this has the potential to exposing the data of users who are not subject to investigation.

F. Data retention

A further point brought up by Kamara and Lauter is that there are no guarantees that on requesting deletion of a file from cloud storage, that it is completely removed from all servers (2010). Kamara and Lauter describes this in the context of an issue relating to regulations on data retention and destruction, but it also an issue if an organisation simply does not want their data to be retained indefinitely in a way that it cannot manage or guarantee destruction.

III. Client-Side Encryption as a key mitigation

Due to all of the above concerns about privacy of data on cloud systems, any static data must not therefore, be stored in plaintext or transmitted across networks unencrypted. Cryptographic keys should also not be stored on the same system as the encrypted data. The key mitigation that solves nearly all these risks when storing static data, is strong client-side encryption, with keys securely stored on the local system or at least away from cloud infrastructure. With data stored on cloud platforms fully encrypted, any breach or compromise of the service is mitigated.

Client-side encryption is not a silver bullet however – encryption can be broken, particularly through improper implementation or use of unsuitable or outdated algorithms. It will also therefore be necessary to understand the different cryptographic methods and the attacks against them.

There are three main requirements of client-side encryption that must be satisfied for such a mitigation to be successful –

1) Confidentiality
2) Authentication
3) Integrity of Data

A. Confidentiality

The crux of confidentiality is that an attacker cannot reasonably obtain, either the full plaintext or even any information about the plaintext from a ciphertext e.g. an encrypted document or file, in a computationally reasonable time.

The simplest of all attacks against confidentiality to understand and attempt, is a simple brute force search of the encryption key. With a key of N bits, there are 2^N possible keys to attempt. This is mitigated with a suitable key size, so that this attack would take too long to be feasible.

The ciphertext should also be sufficiently scrambled in such a way that it is impossible to differentiate from randomly generated binary data. Even if the attacker has multiple plaintext-ciphertext pairs (i.e. known-plaintext attacks), or is able to obtain ciphertexts for any plaintext (i.e. chosen-plaintext attacks), it should not be possible to obtain any information about the plaintext from a ciphertext. An algorithm that achieves this is said to be “semantically secure”. Distinguishing attacks, such as known-plaintext or chosen-plaintext, exploit weaknesses in the algorithm in producing some uniformity or predictability in the output. To avoid this requires a key of suitable length and an algorithm that contains sufficient complexity in the operations that it performs, resulting in no structural uniformity in the output that can be exploited.

There are issues with encryption schemes that only provide confidentiality. For one, it opens the possibility of decryption oracles and chosen-ciphertext attacks. Under confidentiality only schemes, since the binary data is indistinguishable from random bits, decryption can be attempted on any carefully chosen binary data. By analysing the results, information can be gained to rebuild the encryption key and break the cipher. Partly due to this, an encryption scheme must also be *authenticated*.

B. Authentication

Authentication provides assurance that the encrypted data was encrypted with a particular key. Under the assumption that only the sender has access to the encryption key, it is possible to authenticate that the data came from the sender. Authentication also makes it possible to determine if a particular ciphertext has been modified in any way, so provides message integrity. In cloud storage scenarios, where the client does not have full control over the data, it is vital that such message tampering is detected.

Authentication can be provided by use of a Message Authentication Code (MAC) – in one of three possible configurations: Encrypt-and-MAC, MAC-then-Encrypt or Encrypt-then-MAC (Bellare and Namprempre, 2000). In the paper by Bellare and Namprempre, two types of authenticity/integrity are considered: 1. integrity of plaintexts – that it is computationally infeasible to produce an authenticated ciphertext from a plaintext that was not encrypted by the sender, and 2. integrity of ciphertexts that is infeasible to produce any kind of valid ciphertext, even if a previous plaintext is known. The conclusion of the paper is that only Encrypt-then-MAC provides both full privacy and both types of integrity given that a strong, unforgeable MAC is used.

A useful feature of integrity of plaintexts as above is that it means that chosen-ciphertext attacks are no longer possible. An authentication algorithm will recognise when a ciphertext is malformed, that is, it is has not been produced with a valid key or by the correct encryption algorithm and will refuse to decrypt it.

C. Integrity

Authentication and integrity in this context are generally used synonymously, however there exists attacks, namely replay, forking or rollback attacks in which it’s possible to have authentication without integrity (Yun et al., 2009, Yang et al., 2013, Ferguson et al., 2010, p. 26). This is in a situation where there may be different versions of authenticated messages, documents or information. The attacker restores or rolls back a previous, older, but authenticated version of a file or replays or reorders previous messages. The older document or message will be successfully decrypted and authenticated despite being out of date. This introduces a new notion of integrity as a version, time-based or order integrity, different from the integrity of a single file or message. Authenticated Encryption algorithms, or Encryption algorithms with an additional MAC generally only provide integrity of a single message. In order to also ensure version or order integrity, additional metadata about the latest version must be stored in some way.

D. Side channel attacks

Side channel attacks are a category of attack that attacks the particular implementation, rather than the algorithm itself. For example, timing attacks are used by determining the time taken for operations to be completed. Osvik et al. describe how these timing attacks were able to reveal the full AES encryption key used by the dm-crypt filesystem encryption subsystem of Linux in just 65ms, by analysing timings in the CPU memory cache (2005). They also exposed similar timing attacks in the OpenSSL library.

Jungk and Bhasin have shown that by only analysing power consumption or electromagnetic (EM) leaks it is possible to extract encryption keys (2017). Due to the requirement of physical access for this attack, it is generally less relevant for traditional networks.

IV. Existing solutions

A. Native cloud encryption

There are a number of existing solutions for encrypting data on cloud storage.

Cloud providers themselves offer encryption options. For example AWS S3 allows server side encryption, using either AWS managed keys or customer provided keys. While Amazon states that customer provided keys are removed from memory once data has been encrypted (AWS, 2020*c*), there is a fatal flaw in that the key has to be sent to AWS with each upload and download request. For at least the time of encryption and decryption, the key is available plaintext in the server memory. This opens the possibility for many external and internal attacks as above. As all encryption is handled server side, there is no guarantee that files are actually stored encrypted with the provided keys. There also remains the possibility for tampering unless we implement some client-side authentication and integrity checks. Even worse, with AWS managed keys, the encryption keys are stored at rest on AWS servers.

There are a number of cryptographic methods that can provide both proof of storage and proof of retrievability without actually downloading the files (Gorke et al., 2017; Van Dijk et al., 2012; Juels and Kaliski, 2007). However these typically are not possible with pure storage based cloud solutions, e.g. AWS S3, or Google Cloud Storage as they require additional code execution on the cloud server in a challenge-response format.

Amazon does provide a single-tenant, cloud-based hardware security module (CloudHSM) to create, manage and store encryption keys. As this is based on renting a physical, dedicated (single-tenant) hardware appliance that Amazon manages, the cost is understandably prohibitive for most applications (AWS, 2020*a*).

B. *Linux utilities*

There is a solid range of Linux utilities to backup or sync files with cloud storage, however the encryption standards used in these are somewhat lacking. Restic (Neumann, n.d.) provides a snapshot based backup system, and Rclone (Craig-Wood, n.d.*a*) is an rsync like clone tool to sync files to cloud providers.

Rclone uses the NaCL library secretbox implementation, using XSalsa20-Poly1305 for encryption and authentication. The use of XSalsa20-Poly1305 is well tested and peer reviewed (Bernstein, 2008*b*; Ishiguro et al., 2011), however there are some concerns that the implementation in Rclone either leaks information about files and directories, or fails to properly ensure integrity of the data.

The Rclone crypt documentation states ”Hashes are not stored for crypt. However the data integrity is protected by an extremely strong crypto authenticator” (Craig-Wood, n.d.*b*). While Poly1305 is a strong authenticator for a single file, importantly, this approach does guarantee version integrity (as per Integrity section above). Some kind of additional hash or checksum of the latest version, which Rclone specifically does not store, would be required to do so. There is therefore, the potential for an attacker to replace a file with an out of date, but authenticated version.

Additionally, the filename and directory encryption aspect of Rclone leaks information regarding both the file names and directory structure. There are several options for filename and directory name encryption, ranging from no modification at all, a simple obfuscation based on a simple rotate of characters, and the highest security mode in which both filenames and directories are encrypted. Even in the highest security mode, the overall directory and file structure remains the same. This has the potential for some analysis by matching known folder and file structures, potentially resulting in an attacker obtaining some known-plaintext pairs. Additionally, identical filename or directory names encrypt to identical strings, further leaking information about which files and directories have identical names, allowing further analysis of the information stored.

It is also worth mentioning that while Salsa20 used by Rclone is considered secure, Bernstein, the creator of the algorithm has published an improved variant called ChaCha20 in the paper (Bernstein, 2008*a*). Bernstein states it is ”designed to improve diffusion per round, conjecturally increasing resistance to cryptanalysis, while preserving—and often improving—time per round”.

Restic uses AES-256 in counter mode (CTR mode) and authenticates separately using Poly1305-AES. It is designed to be more of an infrequent backup tool, creating snapshots in time of a directory. It uses data blobs to store multiple original files in a single “pack” file, and an index file to track this. This avoids the issue Rclone has with exposing directory and file structure. Additionally each blob is separately hashed and tracked in the index, so any rollback or forking attack should be detected.

The data in Restic is protected behind master keys, derived using scrypt from a user supplied password. All the information required to derive the master keys, with the exception of the actual password, is saved in plaintext on the cloud platform under the key directory; for example the salt used, and the scrypt parameters. At first glance, this appears to be the weakest link in Restic’s security, since only a single password is between an attacker and full decryption of all files. The purpose of using scrypt as a Key-Derivation Function (KDF) however, is specifically to avoid brute force attacks against the key. Key derivation is designed to take significant time and relatively large amounts of memory and resources. Depending on the entropy/security of the password, the parameters used in scrypt and the resources available to an attacker, a brute force attack on the password may be still feasible. In the worst case where a commonly used password is used, a dictionary attack will quickly reveal the password (Restic, n.d.).

C. *Personal cloud storage wrappers*

There are solutions that are designed to encrypt data client-side before uploading to personal cloud storage services like Dropbox and Google Drive. They generally work similar to the native personal cloud sync apps that Dropbox or Google provide, in that cloud storage is mounted as a drive/folder, but they wrap encrypt/decrypt functions around upload/downloads. Boxcryptor is one of these solutions, however this is a closed-source, “blackbox” encryption software that is limited in the free version. No details are provided on the website about what encryption algorithms or standards are used, therefore there is no way to audit the security of this. Potentially worse is that with the default settings, your encryption keys are stored on their servers. This appears to be to make using the software on multiple devices easier, however is a clear security risk (Boxcryptor, n.d.). Cryptomator is similar software, however it is open-source, free to use and appears to have sane and secure defaults (Cryptomator, n.d.). Files and paths are encrypted with AES-256 and stored in a flattened file/directory hierarchy on remote storage. Management of files can be done through a virtual drive using WebDAV. It appears to be a good choice for personal files but support is only for the four main personal cloud providers - iCloud, Google Drive, Dropbox and OneDrive.

V. Requirements capture

The overall goal is to backup and/or sync files between a Linux machine and cloud storage, utilising strong client-side encryption with secure defaults. There should, in general, be no possibility to store files in an insecure way. The utility should be easy to use, providing a high-level interface that abstracts key handling and encryption functions in the background. Significant error checking and handling is also required to prevent data loss.

The program should have the ability to run as daemon, with the ability watch certain files or directories for changes on the local file system in the background. Any changes or new files should be automatically encrypted and uploaded to cloud storage. In the case where a directory with a recursive flag is added, the file system watcher should also add/watch all subdirectories and files recursively as well.

The design should be modular such that any cloud platform that supports an API allowing a user to upload/download can be easily added. In general an Object-Oriented approach is to be taken to allow easy modification/enhancement of the software functions.

A. *Requirements List*

- Utilise strong client-side authenticated encryption with secure defaults. There should, in general, be no possibility to store files in an insecure way.
- Prevent rollback/forking attacks.
- Hide completely filename and directory structure on remote storage.
- Providing an easy to use high-level interface that abstracts key handling, encryption functions and low level operations.
- Daemon service with the ability to watch certain files or directories on the local file system, and automatically upload any changes or new files to cloud storage. There should be a flag to recursively watch directories and it's subdirectories.
- Modular code such that any cloud platform that supports an API allowing upload/downloading can be easily added.
- Performant. Use of multi-threading to complete operations in the background. Encryption should not significantly impair data upload and retrieval, or other functions of the software.

VI. Program design and architecture

A. *Daemon/Client*

As per the requirements, a daemon/service is required to be running in the background constantly. This should be watching for any file changes, and handling the associated encryption/upload processes. Since we want to be able communicate with the daemon, either to tell it to watch a certain directory, or to download a certain file, we need to implement some way of doing so. A common technique for controlling daemons such as webservers is to use a configuration file, however the daemon or config must then be reloaded if any changes are made. Further, while this may work for telling the daemon what directories or files it should watch, this process is unwieldy and would be unsuitable to request a download of certain files from cloud storage. In this case an additional utility is required, either to download the file itself or to communicate to the daemon that it should do so.

[diagram – daemon and thin client]

The preferred solution is to leave all major functions, such as upload/download within the daemon, and implement a client utility to make and pass requests to the daemon. On Linux this type of inter-process communication can be achieved with local or *unix domain* sockets. Similar to TCP/IP sockets for server/client communication through the internet, this allows two processes on the same system to communicate through the operating system kernel.

B. *High level daemon overview*

[diagram – 4 main subsystems and interconnections]

There are 4 main sub-systems required in the daemon to meet the requirements:

- Local filesystem watcher
- Database/Index
- File encryption/decryption
- Upload/download to/from cloud storage

Since the above design has multiple on-going processes in the same program, multi-threading should ensure the code is performant and improve and the usability of the program. Without multi-threading, when uploading or encrypting a large file, this will block all other operations such as watching the filesystem for changes until that operation is complete.

An internal index is required in the daemon to keep track of watched files and directories. Some kind of database will also be required for persistence as well if the Linux host is reset, the program crashes or the daemon is stopped and restarted for any other reason. It is also desired to be able to backup and restore the database from cloud storage in the event of complete local data loss.

The database could be a simple JSON or CSV file, or a more fully featured database.

- Serialise data into a JSON file or CSV, with a large index will take a long time to parse through entire file and amend/change small details. Multiple JSON or CSV files may be required.
- With SQLite, we can quickly add, delete, update and query records. Small blobs can be written and read around 35% than filesystem read/writes using fread() and fwrite() (SQLite, n.d.). The SQLite database is completely self-contained within the program, and does not require an

additional database server process or software to be running, supports ACID transactions, and produces a single database file that can be easily backed up. In addition, it supports online backup so it is possible to backup the database while it is still in use.

As the database file will be continually updated with small reads and writes from anywhere in the database, it is preferred over JSON or CSV files. It should not only be more performant, but it simplifies database operations and provides a simple backup solution to a single file.

C. Programming language choice

With the above high level design features, C++ is a natural choice. To create a Linux daemon, and utilise local sockets these are natural in C/C++, requiring no additional wrappers. Further, most cryptographic libraries are written in C, which can also be directly used in C++ applications without wrappers. The standard library of C++ also contains excellent interaction with the Linux filesystem and has excellent multi-threading support. Generally C++ is more performant and uses less memory than other languages such as Python or Java (Debian, n.d.).

VII. Cryptographic implementation and design

The general rule and accepted guideline regarding using encryption in software development is not to develop your own cryptographic algorithms. In other words, use algorithms that have undergone significant cryptanalysis, testing and development (Schneier, 1998). Developing secure cryptographic algorithms is hard. In the book Cryptography Engineering it is stated that there is no known way of knowing if a cryptographic system is secure. In general, the only way this is done is by publishing the algorithm and have other cryptographers and cryptanalysts review it (Ferguson et al., 2010, p. 13). When choosing the algorithm to use, we should therefore be looking at algorithms that have been extensively peer reviewed. The context of use is also important. What may be a secure algorithm in one context, may have weaknesses in another.

In a similar vein it is preferable to use implementations of algorithms from well-known and tested libraries rather than implement them yourself. Even if the implementation is correct, it is easy to introduce side-channel attacks such as timing attacks that result in a weakening of the algorithm. It still remains that even using well known and tested libraries, there is still the potential to incorrectly and unsafely use some of these in code (Bernstein et al., 2012). For example, if nonces are reused with some algorithms then we "tend to lose confidentiality of messages [. . .] and authenticity tends to collapse completely for all messages" (Langley, 2015). The conclusion of all this is that securely implementing cryptography is not easy. Care must be taken when using libraries and algorithms to ensure they are used properly.

The security of a system should also not rely on obscurity, i.e. by hiding the implementation to be effective. It should be computationally infeasible to break even if the full implementation is known. This is Kerckhoffs's principle – "the security of the encryption scheme must depend only on the secrecy of the key, and not on the secrecy of the algorithm" (Ferguson et al., 2010, p. 24).

There are two main types of encryption to consider. Either asymmetric i.e. public key encryption or symmetric, where a single encryption key is used and shared between parties. When considering file encryption, symmetric key encryption is preferred as it is much less computationally intensive (Njuki et al., 2018). A hybrid method can be used by exchanging symmetric keys via public-key encryption, such as via a Diffie-Hellman exchange.

Either a stream cipher or block cipher in a mode such as Output Feedback Mode (OFB), where the previously computed ciphertext is used as input in future blocks, is preferred to avoid identical plaintext blocks encrypting to identical ciphertext. With identical ciphertext blocks, file analysis is possible between files to identify identical plaintext blocks, leaking information about the file contents (Yun et al., 2009).

A. Deciding on suitable Authenticated-Encryption algorithms

As previously established, Authenticated-Encryption is essential with any cloud based application to prevent tampering of encrypted files (Langley, 2015). This requires an algorithm that combines authentication with encryption, i.e. Authenticated-Encryption (AE) or Authenticated-Encryption with Associated Data (AEAD) algorithms, or alternatively, use a standard, un-authenticated encryption mode with a MAC in the Encrypt-then-MAC format. Generally AEs and AEADs operate faster over large amounts of data, since a separate MAC takes extra time to compute, especially if the MAC is suitably secure.

There are also practical considerations when encrypting files, particularly for encrypting large files. Generally speaking to encrypt large files it will not be possible, or desirable to hold an entire copy of the file in memory, so it must be possible to stream the contents of the file to the encryption algorithm, or split the file into chunks to encrypt separately.

This brings up an issue relating to scenarios where ordering is important, as described by Langley (2015). Splitting a large file into chunks will require establishing and checking the order or chunks, so messages cannot be recomposed in the wrong order or truncated without failing. As Langley states, there are no existing standards or practices for this, so a self-developed implementation has the possibility for being insecure.

The most popular AEAD algorithms are AES in EAX, CCM and GCM modes, and Salsa20/ChaCha20-Poly1305. AES-GCM is the fastest of the AES variants, however it has the limitation that only roughly 350GB of data can be encrypted with a single key (Libsodium, n.d.*a*). Further, the nonces used are short. While this mode is used extensively in TLS, it is not particularly suitable for file encryption. A custom chunking implementation would need to be written to resolve the issue of chunk re-ordering as above.

Salsa20 and the newer ChaCha20 variant can be coupled with the Poly1305 authenticator. These were created by Daniel Bernstein and were described in a series of papers (Bernstein, 2005*a*, 2008*b*, 2005*b*, 2008*a*; Nir et al., 2015). Bernstein has further extended these ciphers by adding 'X' variants, i.e. XSalsa20 and XChaCha20 in which the nonce is extended. A useful feature of this for file encryption is that it allows an unlimited amount of data to be encrypted without becoming insecure (2011). Bernstein has also stated that Salsa20 has many benefits over AES, generally being faster and more secure (2012). In addition, the Poly1305 authenticator is guaranteed to be as secure as the algorithm it is paired with (2005*b*).

B. Choosing the correct encryption library

There are a large number of encryption libraries and algorithms to choose from. Three of the most established C and C++ libraries are the OpenSSL library, Crypto++ and NaCL (in particular libsodium).

OpenSSL is FIPS 140-2 validated – a U.S. government security standard. The Authenticated-Encryption with Associated Data (AEAD) schemes supported are AES in EAX, CCM and GCM modes. As these modes do not support splitting large files into chunks this will have to be handled separately.

Crypto++ supported AEADs are AES in GCM, CCM and EAX modes, ChaCha20Poly1305 and XChaCha20Poly1305.

The library chosen for this software is NaCL, created by Daniel Bernstein, author of the Salsa20/ChaCha20 ciphers and Poly1305 authenticator. The motivation behind the library is explained in Bernstein et al. (2012) – to create a simple, high-level interface, abstracting low level cryptographic functionality to avoid implementation errors. It is designed such that there are no low-security options or defaults, and authenticated encryption is implemented in such a way that unauthenticated ciphertext is never decrypted, which is known to cause some issues (Langley, 2015). In addition, timing-attacks are mitigated. Importantly for the purposes of this software, there is a streaming implementation using XChaCha20-Poly1305 such that messages cannot be "truncated, removed, reordered, duplicated or modified without this being detected" (Libsodium, n.d.*b*), in response to the concerns of Langley (2014). In addition, "the same sequence encrypted twice will produce different ciphertexts" (Libsodium, n.d.*b*) to avoid file/cryptanalysis. As the library was created by the author of the cryptographic algorithms used, we have some confidence the implementation is secure.

C. Filename encryption

For full confidentiality, encrypted files will need to be stored on cloud storage with hidden file names and paths. Both the directory structure, and filenames will leak information about the stored files so the ideal scenario is a completely flat file structure on the cloud with encrypted or otherwise obfuscated filenames that can be restored when downloading.

One option is to encrypt the path and filename with a secret key, then encode with base64 or base32 into a filename compatible string. A suitable algorithm must be chosen such that different nonces are used and stored with the encrypted string, otherwise identical filenames will encrypt to identical paths, leaking information about files that share a filename. As a benefit this allows the possibility to restore files without the need for an index.

Alternatively the path/filename or file can be hashed with a salt value. A unique salt value per file would be required otherwise identical filenames or files would hash to identical filenames. This would leak information as above, and additionally open the possibility for an attack via rainbow tables by pre-computing hashes to all possible filenames. A fixed, or short salt would allow an attacker to compute a rainbow table for every possible salt and filename, so in order for this to be computationally infeasible, a long, unique hash per file is required. As the hash is a one-way function, an index file would be required to track which hashes map to which file and also store the salt separately. The hash could also be used as a checksum and offer protection against rollback attacks (see integrity above), and assuming we have a list of the filenames with the salts used we can recreate the names of encrypted files.

The most secure option would be to store files under completely random strings that have no connection to original filename or file. This removes the possibility of cryptanalysis of the filenames. In this option an index is required/vital and must be kept safe and securely backed up. Corruption or loss of the index would result in being unable to restore the original filenames/directory structure, however this is also the case with hashed filenames. As such this is the preferred option over hashes as there is no connection between the filename of the encrypted data and the original file. In both instances, an index file is required. Further, it is possible to backup the index file/database to cloud storage in a secure way. As long as the master encryption key is retained, it will be possible to restore the database from remote storage.

D. Index backup

As above, loss of the index file would result in there being no way to restore filenames and directory structure from the encrypted files on remote storage. The solution to this is to backup the index file to cloud storage, however this presents several issues. It is not possible to resolve the index backup filename to a random string, since this requires the index file itself and we are assuming complete loss of the index. If the index filename is not hidden it may present an attacker with a known-plaintext pair, especially in the case where a default empty database file is uploaded in the first instance, before any file or directory watches have been added.

The solution is to derive the filename of the index backup in a way that is reproducible or verifiable with the master encryption key. The filename should be indistinguishable from the randomly generated filenames of regular encrypted files. In addition, in order to avoid a timing-attack and providing a possible known-plaintext pair to an attacker, the database will not be backed up when it is empty.

The NaCL libsodium library used for file encryption provides a secure way to do this via their key derivation function API (crypto_kdf) which uses the Blake2B algorithm internally. From the master key, it's possible to derive a number of subkeys in a deterministic way, such that given a subkey there is no way to compute back to the master key (Libsodium, n.d.*c*). To add an even stronger layer of security in case a weakness is found in the key derivation function, the subkey is base64 encoded and used as a password in libsodium's Argon2id password based key derivation function (PBKDF), to generate a new key with a random nonce (Libsodium, n.d.*d*). Both the new key and random nonce are base64 URL encoded and concatenated and used as the filename to backup the index. In reverse, to determine which filenames on cloud storage represent index files, we can deterministically derive the same subkey from the master key; then, for each filename on the cloud, the derived subkey is used as a password while providing the last 24 chars of the filename as the nonce. If the base64 output of the derived key matches the first 64 chars of the filename, we have verification that this is an index file backup created with the above process. While this technique is not particularly fast to complete with many files, such an index restoration is likely to be an extremely infrequent event.

The index backup process also produces an unencrypted index.backup file locally alongside the normal index. This can be locally backed up via other means if remote backup of the index is not required.

[diagram: master key $\rightarrow$ subkey $\rightarrow$ password KDF function + nonce]

VIII. Local security mitigations

This software assumes that the host local system is secure, however some local risks are considered. Unix filesystem permissions are used to restrict access to certain critical parts, for example the encryption key, database and local socket used to communicate with the daemon. However, in general there is a reliance on the user to ensure the local system is sufficiently secure and up to date.

IX. Testing and benchmarks

Todo? Hard to give benchmarks not based solely off my computer/internet speed. Encryption algorithm benchmarks have already been provided in references. Could test software with thousands of files/large sizes and report that it works.

X. Critical analysis

The original goal was to ensure full confidentiality, authentication and integrity of data on cloud storage. To this end great care has been taken to avoid any possible information leaks from uploaded data. The implementation of storing files on the cloud in a flat file structure with randomly generated filenames guarantees that no information can be extracted from this. A strong, well-reviewed and tested authenticated encryption scheme, with additional hash checks/storage of the latest version in the index prevents undetected tampering of the files on cloud storage.

In order to ensure complete security of data, there is some computational and network bandwidth overhead. For example, should a file change by a single bit, the entire file will be re-encrypted and uploaded in full. The alternative is to use a block cipher such so encrypted blocks are completely independent, and only updated plaintext blocks can be encrypted and uploaded. Houti and Miele (2015) state this results in a number of security and confidentiality concerns where data is stored in an unsecure location, such as on the cloud. They provide a potential solution to this, by creating a local index/mapping to authenticate and verify integrity of these updated blocks. This may be an area to research further for future versions. Restic solves this problem by uploading diff files, and recreating files on download Restic (n.d.).

Further overhead is created in backing up the index to the cloud. Depending on the amount of tracked files, the index may become extremely large, resulting in a somewhat significant cost of encryption and uploading. If tracking of an extremely large number of files is required, this can be resolved either by turning off remote index backup and implementing a local backup solution, or reducing the frequency of backup operations. In most instances however the index will not be large enough to cause concern.

There is always a cost/benefit calculation in determining the level of security provided. Since the goal of the software is to provide full confidentiality and authentication, some overhead is unavoidable. Despite this, by using a multi-threaded approach, the software remains performant and responsive by splitting operations into separate threads. There are still a number of improvements that can be made in this area. Currently the upload/download and encryption/decryption queue processes items in a linear fashion; performance of this can be improved by parallelising multiple files to be processed at once.

There is still a potential timing attack that may allow an attacker to determine which file is the encrypted index backup, as long as they are able to continually monitor the remote storage. As the index is backed up at regular intervals, it should be possible to determine which file is updated the most/at regular intervals. There is also currently no way to hide the size of encrypted files, so by monitoring files which are regularly updating and increase in size in proportion to new uploads, it should be possible to determine which is the index. Although the software is careful not to upload a blank database, and therefore providing a known-plaintext pair, it would still be preferable to hide this if possible. Currently there are no known attacks that would break the full Salsa20 or ChaCha20 variants even with known-plaintext pairs, with the best attacks breaking only 7 rounds of 20 (Salsa et al., 2008; Ishiguro et al., 2011).

Currently there is no way to support rollback protection on database/index backup file. For regular files, a separate file hash/checksum is stored in the index to mitigate this. However, in the case of complete local data loss, there is no way to store this hash. This issue is somewhat mitigated if the index backup is indistinguishable in name from other files on cloud storage. If an attacker is not aware of what file is the index, or that an index is used, they may rollback multiple files which will be detected and a warning is displayed to the user that data tampering has been detected. However due to the above timing attacks, there may be scenarios where the index file can be determined. If the index file is tampered with or rolled back to a previous version, the only consequence is likely to be data loss, so a regular local backup of the index is also recommended. It may be possible to encode a checksum within the filename to resolve this issue, but this requires further testing/work.

A. Limitations

The current software only supports encrypting files with a single master encryption key. Typically cloud storage is utilised as a way to share files between multiple parties. There are multi-user encryption schemes that can provide and manage access between multiple parties, such as attribute based encryption (Kamara and Lauter, 2010; Zhang et al., 2013).

The software is also somewhat limited in that it is designed only for use with static data. If any computations are required on the data then it is not possible without downloading the data in full and decrypting. There are several homomorphic encryption techniques that allow processing on encrypting data, however to date they are limited in the computations that can be completed (Alloghani et al., 2019; Kucherov et al., 2020). Since these approaches have to be somewhat specialised, they are not particularly suitable for a general secure data storage solution and are outside the scope of an application such as this.

XI. Future work/improvements

The file system watcher currently uses the std::filesystem from the C++ standard library to regularly poll the filesystem and monitor for new files/directories, deleted files or changes in modification time. A system such as this is necessary in order to detect changes that have occurred when the daemon was not running. However, there is a subsystem within the Linux kernel, the *inotify* API which can notify an application of any changes to files or directories (Free Software Foundation, 2020). This would reduce the cost of regular polling via the std::filesystem library, which could be significant if the application is watching a large amount of files and directories. The std::filesystem method would still be retained and run on initial start-up of the daemon, then handing off to receive notifications from inotify while the daemon is running.

The current implementation of file encryption and upload is to encrypt to a temporary file on local storage before uploading. This can be improved by streaming the encryption stream directly to remote storage. With the current NaCL secretstream implementation this should be possible with many cloud storage providers such as AWS S3.

The handling of available files on remote storage can also be improved. Currently the *enclone –list remote* command line argument returns a list of available files and versions that is somewhat awkward with large amounts of files. This could be improved by outputting a formatted tree structure instead of a linear list. There is also the option to mount a remote drive on Linux with *fusermount*, and wrap encryption/decryption functions around this. RClone contains an excellent working implementation of this (Craig-Wood, n.d.*c*).

Kamara and Lauter (2010) describe searchable encryption schemes that can be used to save encrypted indexes on remote storage in such a way that it can be queried without downloading and encrypting it. This provides an interesting area for further research but is limited in its application on pure cloud storage providers such as AWS S3, requiring a separate cloud service allowing remote code execution. Similar issues exist with implementing Proof of Storage (Gorke et al., 2017; Juels and Kaliski, 2007).

References

Aiello, B., Warfield, A., Nanavati, M. and Colp, P. (2014), 'Cloud Security - A Gathering Storm', *Communications of the ACM* **57**(5), 70–79.

Alani, M. M. (2017), 'Prioritizing cloud security controls', *ACM International Conference Proceeding Series* .

Alloghani, M., M. Alani, M., Al-Jumeily, D., Baker, T., Mustafina, J., Hussain, A. and J. Aljaaf, A. (2019), 'A systematic review on the status and progress of homomorphic encryption technologies', *Journal of Information Security and Applications* **48**, 1–10.

AWS (2020*a*), 'AWS CloudHSM'.
URL: *https://aws.amazon.com/cloudhsm/*

AWS (2020*b*), 'General S3 FAQs'.
URL: *https://aws.amazon.com/s3/faqs/*

AWS (2020*c*), 'Protecting data using server-side encryption with customer-provided encryption keys'.
URL: *https://docs.aws.amazon.com/AmazonS3/latest/dev/ServerSideEncryptionCustomerKeys.html*

Bellare, M. and Namprempre, C. (2000), 'Authenticated encryption: Relations among notions and analysis of the generic composition paradigm', *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* **1976**, 517–530.

Bernstein, D. J. (2005*a*), 'Salsa20 design'.

Bernstein, D. J. (2005*b*), 'The poly1305-AES message-authentication code', *Lecture Notes in Computer Science* **3557**, 32–49.

Bernstein, D. J. (2008*a*), 'ChaCha, a variant of Salsa20', *Workshop Record of SASC* pp. 1–6.
URL: *http://cr.yp.to/chacha/chacha-20080120.pdf*

Bernstein, D. J. (2008*b*), 'The salsa20 family of stream ciphers', *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* **4986 LNCS**, 84–97.

Bernstein, D. J. (2011), 'Extending the Salsa20 nonce', (Mc 152), 1 – 14.
URL: *http://cr.yp.to/snuffle/xsalsa-20110204.pdf*

Bernstein, D. J., Lange, T. and Schwabe, P. (2012), 'The security impact of a new cryptographic library', *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* **7533 LNCS**, 159–176.

Boxcryptor (n.d.), 'Export Your Keys'.
URL: *https://www.boxcryptor.com/en/help/boxcryptor-account/windows/#export-your-keys*

Craig-Wood, N. (n.d.*a*), 'rclone'.
URL: *rclone.org*

Craig-Wood, N. (n.d.*b*), 'rclone: Crypt'.
URL: *https://rclone.org/crypt/*

Craig-Wood, N. (n.d.*c*), 'rclone: Mount'.
URL: *https://rclone.org/commands/rclone_mount/*

Cryptomator (n.d.), 'Security Architecture'.
URL: *https://docs.cryptomator.org/en/latest/security/architecture/*

Debian (n.d.), 'The Computer Language Benchmarks Game'.
URL: *https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html*

Dropbox (2019), 'Terms of Service'.
URL: *https://www.dropbox.com/en_GB/terms*

Dropbox (n.d.), 'FAQ: Is Dropbox safe to use?'.
URL: *https://help.dropbox.com/accounts-billing/security/safe-to-use*

El Makkaoui, K., Ezzati, A., Beni-Hssane, A. and Motamed, C. (2017), 'Cloud security and privacy model for providing secure cloud services', *Proceedings of 2016 International Conference on Cloud Computing Technologies and Applications, CloudTech 2016* pp. 81–86.

Ferguson, N., Schneier, B. and Kohno, T. (2010), *Cryptography Engineering: Design Principles and Practical Applications*, Wiley Publishing, Inc.

Free Software Foundation (2020), 'inotify(7) — Linux manual page'.
URL: *https://man7.org/linux/man-pages/man7/inotify.7.html*

Geffner, J. (2015), 'VENOM: Virtualized Environment Neglected Operations Manipulation'.
URL: *https://venom.crowdstrike.com/*

Gorke, C. A., Armknecht, F., Janson, C. and Cid, C. (2017), 'Cloud storage file recoverability', *SCC 2017 - Proceedings of the 5th ACM International Workshop on Security in Cloud Computing, co-located with ASIA CCS 2017* pp. 19–26.

Harnik, D., Pinkas, B. and Shulman-Peleg, A. (2010), 'Side channels in cloud services: Deduplication in cloud storage', *IEEE*

Security and Privacy **8**(6), 40–47.

Houti, Y. E. and Miele, A. (2015), 'Efficient Update of Encrypted Files for Cloud Storage', *Proceedings - 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing, UCC 2015* pp. 565–570.

Ishiguro, T., Kiyomoto, S. and Miyake, Y. (2011), 'Latin dances revisited: New analytic results of Salsa20 and ChaCha', *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* **7043 LNCS**, 255–266.

Juels, A. and Kaliski, B. S. (2007), 'PORs: Proofs of retrievability for large files', *Proceedings of the ACM Conference on Computer and Communications Security* pp. 584–597.

Jungk, B. and Bhasin, S. (2017), 'Don't fall into a trap: Physical side-channel analysis of ChaCha20-Poly1305', *Proceedings of the 2017 Design, Automation and Test in Europe, DATE 2017* pp. 1110–1115.

Kamara, S. and Lauter, K. (2010), 'Cryptographic cloud storage', *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* **6054 LNCS**, 136–149.

Kandias, M., Virvilis, N. and Gritzalis, D. (2007), 'The Insider Threat in Cloud Computing', *Critical Information Infrastructures* (c), 61–76.

Kandukuri, B. R., Ramakrishna, P. V. and Rakshit, A. (2009), 'Cloud security issues', *SCC 2009 - 2009 IEEE International Conference on Services Computing* pp. 517–520.

Kaufman, L. M. (2009), 'Data security in the world of cloud computing', *IEEE Security and Privacy* **7**(4), 61–64.

Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M. and Yarom, Y. (2020), 'Spectre attacks: Exploiting Speculative Execution', *Communications of the ACM* **63**(7), 93–101.

Kucherov, N. N., Deryabin, M. A. and Babenko, M. G. (2020), 'Homomorphic Encryption Methods Review', *Proceedings of the 2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering, EIConRus 2020* (19), 370–373.

Langley, A. (2014), 'Encrypting streams'.
URL: *https://www.imperialviolet.org/2014/06/27/streamingencryption.html*

Langley, A. (2015), 'AEADs: getting better at symmetric cryptography'.
URL: *https://www.imperialviolet.org/2015/05/16/aeads.html*

Libsodium (n.d.*a*), 'AES256-GCM'.
URL: *https://libsodium.gitbook.io/doc/secret-key_cryptography/aead/aes-256-gcm*

Libsodium (n.d.*b*), 'Encrypted streams and file encryption'.
URL: *https://libsodium.gitbook.io/doc/secret-key_cryptography/secretstream*

Libsodium (n.d.*c*), 'Key derivation'.
URL: *https://libsodium.gitbook.io/doc/key_derivation*

Libsodium (n.d.*d*), 'Password hashing'.
URL: *https://libsodium.gitbook.io/doc/password_hashing*

Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y. and Hamburg, M. (2018), 'Meltdown: Reading kernel memory from user space', *Proceedings of the 27th USENIX Security Symposium* pp. 973–990.

Neumann, A. (n.d.), 'Restic'.
URL: *https://restic.net*

Nir, Y., Check Point, Langley, A. and Google, I. (2015), 'RFC7539: ChaCha20 and Poly1305 for IETF Protocols'.
URL: *https://tools.ietf.org/html/rfc7539*

Njuki, S., Zhang, J., Too, E. C. and Richard, R. (2018), 'An evaluation on securing cloud systems based on cryptographic key algorithms', *ACM International Conference Proceeding Series* pp. 14–20.

Osvik, D., Shamir, A. and Tromer, E. (2005), 'Cache attacks and countermeasures: the case of AES (Extended Version)', *Topics in Cryptology–CT-RSA 2005* pp. 1–25.
URL: *http://link.springer.com/chapter/10.1007/11605805_1*

Restic (n.d.), 'Restic Documentation: References'.
URL: *https://restic.readthedocs.io/en/latest/100_references.html*

Salsa, A., Aumasson, J.-p., Fischer, S., Khazaei, S., Meier, W. and Rechberger, C. (2008), 'New Features of Latin Dances :', *Fast Software …* .
URL: *http://link.springer.com/chapter/10.1007/978-3-540-71039-4_30*

Schneier, B. (1998), 'Memo to the Amateur Cipher Designer'.
URL: *https://www.schneier.com/crypto-gram/archives/1998/1015.html#cipherdesign*

SQLite (n.d.), '35% Faster Than The Filesystem'.
URL: *https://www.sqlite.org/fasterthanfs.html*

Sun, X. (2018), 'Critical Security Issues in Cloud Computing: A Survey', *Proceedings - 4th IEEE International Conference* *on Big Data Security on Cloud, BigDataSecurity 2018, 4th IEEE International Conference on High Performance and Smart Computing, HPSC 2018 and 3rd IEEE International Conference on Intelligent Data and Securit* (1), 216–221.

Synopsys Inc (n.d.), 'The Heartbleed Bug'.
URL: *https://heartbleed.com/*

Van Dijk, M., Juels, A., Oprea, A., Rivest, R. L., Stefanov, E. and Triandopoulos, N. (2012), 'Hourglass schemes: How to prove that cloud files are encrypted', *Proceedings of the ACM Conference on Computer and Communications Security* pp. 265–280.

Yan, L., Hao, X., Cheng, Z. and Zhou, R. (2018), 'Cloud Computing Security and Privacy', *Cloud Computing Security and Privacy* pp. 119–123.

Yang, H. J., Costan, V., Zeldovich, N. and Devadas, S. (2013), 'Authenticated storage using small trusted hardware', *Proceedings of the ACM Conference on Computer and Communications Security* pp. 35–46.

Yun, A., Shi, C. and Kim, Y. (2009), 'On protecting integrity and confidentiality of cryptographic file system for outsourced storage', *Proceedings of the ACM Conference on Computer and Communications Security* pp. 67–75.

Zhang, Y., Jia, Z. and Wamg, S. (2013), 'A multi-user searchable symmetric encryption scheme for cloud storage system', *Proceedings - 5th International Conference on Intelligent Networking and Collaborative Systems, INCoS 2013* pp. 815–820.

Zhang, Y., Juels, A., Reiter, M. K. and Ristenpart, T. (2012), 'Cross-VM side channels and their use to extract private keys', *Proceedings of the ACM Conference on Computer and Communications Security* pp. 305–316.