“ALEXANDRU IOAN CUZA” UNIVERSITY FROM IAȘI

FACULTATY OF INFORMATICS

DISSERTATION THESIS

Comparison between cryptographic cloud storage systems

proposed by

Lăzăroi Darius-Marian

Session: *july, 2021*

Scientific coordinator

Prof. Dr. Țiplea Ferucio Laurențiu

“ALEXANDRU IOAN CUZA” UNIVERSITY FROM IAȘI

FACULTATY OF INFORMATICS

Comparison between cryptographic cloud storage systems

Lăzăroi Darius-Marian

Session: *july, 2021*

Scientific coordinator

Prof. Dr. Țiplea Ferucio Laurențiu

Table of Contents

1 Introduction

Cryptographic cloud storage systems and cloud services in general, have gained a lot of popularity in the recent years because of their accessibility and availability. However, the present systems are still not perfect as they require a high level of trust from either the provider, the client or both. For example, providers usually retain their ability to view the stored files in order to make sure that they do not break any rules in the EULA, and some providers even retain their ability to share stored data. Generally, when the provider has access to view the data, it would not be difficult to also create a copy of the data, making this would count as abuse. This would be very hard for an average client to prove as they have no possible way of acquiring any related information to such allegations. Nowadays, this problem has only become more apparent due to the increased amount of machine learning performed on personal user data with the purpose of either customizing ads, or in the more extreme scenarios, manipulation of public opinion. Cloud storage services are the perfect place for obtaining such information, either personal or not, due to the large number of clients and files stored.

One simple solution that can be used by the user to retain complete privacy is to add another layer of encryption locally. This way, the cloud service will not be able to decrypt the files. This also comes with the downside of performing manual key management, which becomes more troublesome as the number of stored files increases.

In the second part, certain theoretical concepts that appear later in the paper are explained.

In the third part, this paper attempts to analyze and compare the security of some popular existing cryptographic cloud storage systems to better understand common practices and to find potential issues. The surveyed providers are: Tresorit, Microsoft Azure, Google Cloud and Amazon S3. The infrastructure security of all 4 of these (Tresorit uses the Microsoft Azure Infrastructure) is very similar. The “Encryption at rest” chapters contain the most differences, as every provider either prioritizes security, speed, or a mix of both with regards to storage security and file access speeds. Here, Tresorit has a complete different approach from the other 3 providers by using a storage system named “Tresorium” and a tree-like lockbox structure.

Service specific documentation (alongside some patents and related scientific articles) is the basis of this comparison. However, some providers either do not elaborate certain concepts/operations (such as authentication in Google Cloud) or they provide a limited amount of details, leaving room for interpretation.

In the final part, we propose a new scheme for cryptographic cloud storage whose purpose is to limit potential abuse by informing the user when a file inspection takes place using AES-256 CBC and Cocks IBE. Cocks IBE will be used as a cryptographic substitute for access control lists, and AES-256 will be used for its IV properties (such as completely changing the ciphertext if it is modified, and block corruption if the same IV that was used at encryption is not used for decryption). Authentication will also be zero-knowledge using Fiat-Shamir. Such a system could even be used by real life providers, as it would inspire additional trust in the service.

2 Theoretical Background

2.1 Hash Functions

A hash function is a process which transforms any dataset into a fixed length character series, regardless of the input. The optimal properties of hash functions are [1]:

- The output must be deterministic, meaning that if the hash function is applied over the same input, it will have the same output every time.
- It must be computationally hard to reverse a hash function.
- The slightest change must lead to a completely different result (if one bit is change it must lead to a completely different hash).
- Finding 2 messages that have the same output must be extremely difficult.
- All possible outputs must have the same probability of occurrence.
- Efficiency.

In general hash functions involve splitting the data into multiple blocks after which the hashing algorithm is applied over multiple rounds like a block cipher. The process is repeated as many times as it is required.[1]

Popular hashing functions include:

- Message Digest (MD)
- Secure Hash Function (SHA)
- RACE Integrity Primitives Evaluation Message Digest (RIPEMD)
- Whirlpool

One of the most popular uses for hash functions is for storing passwords in databases in order to avoid storing it in a clear format for 2 reasons. One being the customers privacy. If the password was stored in a clear format in the database, someone who has access to it could easily log into any account. The second reason is to protect and diminish the effect of possible database breaches. A breach into a database that stores passwords in a clear format would allow access over any account stored in it, however, if the passwords are hashed then the attacker can still access other details. Depending on the contents, the effect can range from mildly problematic to negligible. In a worst-case scenario, an attacker could extract personal data such as names, addresses, phone numbers or other information. In a best-case scenario the only thing compromised is the email address and/or the account name. Such a breach happened to Yahoo and because they stored passwords in plaintext, they have suffered multiple lawsuits, on top of losing many customers and customer credibility.

Another popular use is data integrity check. Because of the way hash functions work, they can be applied over the contents of a message and sent alongside with it. When the message is received, the user can reapply the function over the contents and check if it is the same. If the result is positive then the data was not changed, if it is not, then it is possible a third party tried to change the contents of the message, or an error occurred during transmission.

2.2 Secure Hashing Algorithms (SHA)

Secure Hashing algorithms (in short SHA) are a family of hash functions that produce a fixed size string that looks nothing like the original.

SHA-0 is the original version of the 160-bit hash function published in 1993. It was shortly replaced with SHA-1 due to significant flaws.

SHA-1. When used over a message with the length less than 2^{64} the SHA-1 produces a 160-bit output called message digest. It is called secure because it is computationally unfeasible to find an input string which corresponds to a given message digest or to find two different messages which produce the same digest.[2]

In SHA-1 the input is split into blocks of 512-bit size. After which every block is padded following 3 steps [2]:

- 1 is added to the end of the message.
- 0's are appended until the block length reaches 448-bits and transform the result into hex
- Compute the length of the input text L, split L into 2 blocks of 32 bits ($l_1|l_2 = L$), and then append l_1 and l_2 ,in hexadecimal, at the end of the block.

After that, a series of constants and functions are applied as follows [2]:

1. Divide the current block into 16 32-bit (or byte) segments (resulting vector is named W)
2. A=H0; B=H1; C=H2; D=H3, E=H4
3. *For t=0 to 79 do:*

   S= t AND MASK

   If(T>=16)

   W[S]=S^1(W[(S+13) AND MASK] XOR W[(S+8) AND MASK] XOR W[(s+2) AND MASK] XOR W[S];

   TEMP = S^A(A) + f(t;B,C,D) + E + W[S] + K(t);

   E=D; D=C; C=S^30(B); B=A; A=TEMP
4. H0=H0+A; H1=H1+B; H2=H2+C; H3=H3+D; H4=H4+E;

Where initially:

$$Ht\begin{cases}67452301 & |\, t=0\\ EFCDAB89 & |\, t=1\\ 98BADCFE & |\, t=2\\ 10325476 & |\, t=3\\ C3D2E1F0 & |\, t=4\end{cases} \qquad K(t)=\begin{cases}5A827999 & |0\le t\le 19\\ 6ED9EBA1 & |20\le t\le 39\\ 8F1BBCDC & |40\le t\le 59\\ CA62C1D6 & |60\le t\le 79\end{cases}$$

$$f(t,B,C,D)=\begin{cases}(B\ AND\ C)OR\ \big((NOT\ B)AND\ D\big) & |0\le t\le 19\\ B\ XOR\ C\ XOR\ D & |20\le t\le 39\\ (B\ AND\ C)OR\ (B\ AND\ D)OR\ (C\ AND\ D) & |40\le t\le 59\\ B\ XOR\ C\ XOR\ D & |60\le t\le 79\end{cases}$$

However, SHA-1 is no longer considered secure.

SHA-2 (SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256) is a set of cryptographic hash functions that were first published in 2001. It retains the same structure as the SHA-1 but uses different shifts amounts and different constants.

SHA-3 is the newest version of SHA that follows a different structure than SHA-2, it was first known as Keccak. It was meant to be a replacement for SHA-2.

2.3 Message authentication codes

A message authentication code (MAC), also known as message integrity code (MIC), is a segment of data used to verify if the message came from the supposed sender and whether it was modified during the transmission.

One popular use for MACs is when two entities have already established a secret key between them, and they wish to verify the information sent or received between them. [3]

2.4 Hash-based message authentication code

A HMAC is a specific type of message authentication code that uses a cryptographic hash function and a secret retaining all the properties of a basic MAC.

HMACs are flexible and they can be combined with any iterated cryptographic hash function. They can be viewed as a keyed hash function, where the same key is needed to obtain the initial result.[3]

The authors of [3] state that the goals behind a HMAC are the following:

- To easily use already existing hash functions without modifying them.
- To facilitate key management and key usage.
- To allow for easy replaceability of the hash function used in case a more efficient one is found or if the hash function used is compromised.

The simple HMAC algorithm is [3]:

$$H((K\ XOR\ opad)|H\big((K\ XOR\ ipad)|text\big))$$

Where:

$$H\ is\ the\ hashing\ algorithm$$

$$K\ is\ the\ shared\ key$$

$$B\ is\ the\ byte\ lenth\ of\ the\ block\ used\ by\ the\ hash\ function$$

$$ipad = 0x36\ repeated\ B\ times$$

$$opad = 0x5C\ repeated\ B\ times$$

Here "|" represents concatenation.

The size of K will be normalized by adding 0 at the end until it is of B length.

They key size is irrelevant as the hash function can be applied before usage. An increased key size will not improve the security by much unless the key randomness is considered weak. Choosing a key with a size shorter than the output of the hashing algorithm is "strongly discouraged" by the authors of [3] because it decreases the overall security. Keys must be random and must be refreshed very often. Generally,

refreshing the key very often helps in reducing the damage in case it was compromised. It also gives the attacker shorter available periods of analyzing the algorithm.[3]

The security of a HMAC depends on the hashing algorithm chosen.

2.5 The Transport Layer Security Protocol

The TLS protocol has multiple objectives, the main one consisting of creating a secure channel where two entities can communicate in a safe way.[4] It has two main components: the TLS handshake protocol and the TLS record protocol.[4] The TLS handshake protocol will be wrapped using the TLS record protocol. The TLS record protocol guarantees a secure connection with two basic properties [4]:

- Privacy.
- Reliability.

The TLS handshake protocol is used for authentication alongside negotiation of other cryptographic details such as the encryption/integrity algorithms used or keys.[4]

The TLS handshake protocol provides three properties [4]:

- Secure negotiation.
- Reliable negotiation.
- Authentication using asymmetric keys.

The TLS Record Protocol has the following steps[4][4]:

- First, it splits the data into multiple segments.
- Then it uses a MAC function with one segment and the established key as input.
- Then data is concatenated with the MAC result and afterwards encrypted.
- As a final step the data is sent to the other entity.

The recipient applies all these steps in a reverse order, however, in version 1.2 the decryption was done before the MAC verification which could lead to possible problems. In version 1.3 renegotiations are also forbidden.[4][5]

The TLS handshake has 3 stages: Key exchange, Establishment of other server parameters and Authentication (it is mandatory to authenticate the server but optional to authenticate the client).

The basic TLS handshake in version 1.3 works as follows [4]:

- Key exchange phase
    1. The client sends the ClientHello message containing a random nonce, its protocol versions, a list of available symmetric ciphers and either a set of Diffie-Hellman key shares or a set of pre-shared label keys or both (and other extensions if needed)
    2. The server processes the ClientHello and determines the suitable algorithms and parameters that will be used from now on. It then responds with its own ServerHello which

indicates the accepted connection algorithms and parameters. The combination of ClientHello and ServerHello determines the shared keys.

- Establishment of other server parameters
    1. The server sends the Encrypted Extensions message which is a response to the ClientHello extensions that are not required to determine the cryptographic parameters.
    2. The server sends the Certificate Request message if the certificate-based client authentication is desired. Otherwise this message is omitted.
- Authentication. Here TLS uses the same set of messages every time certificate-based authentication is needed. (Pre-Shared Key based authentication happens naturally as a side effect of key exchange)
    1. First message sent is called Certificate and it represents the certificate of the endpoint and any other extensions. This message is omitted if the client is not authenticating with the certificates. If the server uses some sort of long-term key (asymmetric key pairs) this message will contain some other value corresponding to it.
    2. Next message is CertificateVerify which is a signature over the entire handshake using the private key corresponding to the public key in the certificate message. This message is also omitted if the authentication chosen does not use certificates.
    3. Final message is Finished. A MAC over the entire handshake to provide key confirmation and binding it to the endpoint's identity. In Pre-shared key mode, it also authenticates the handshake.

Upon receiving the final message, the client responds with its own Authentication messages, if requested, and then the authentication finished

2.6 Digital signatures

A digital signature is a construct used to verify the authenticity and integrity of messages. Today the most popular signing algorithm is RSA due to the ease of use and security provided. With RSA, the sender uses his private key to sign the entire message and then the receiver uses his public key to verify it. Because of the way a public key and a private key works, only one person may know the private key, but anyone can know the public key, only one user is able to sign the message but anyone with the public key can verify the signature. Other popular algorithms include DSA (Digital Signing Algorithm) and Elliptic Curve DSA.[32]

Generally, there are 4 steps required in such a scheme:

1. Key generation
2. Key distribution
3. Signing
4. Signature verifying

2.7 Cryptographic certificates

A digital certificate, also known as a cryptographic certificate or a public-key certificate, is a document used to prove authenticity over an identity, granted by a higher authority. Some common fields in such certificates include serial numbers, subject, issuer, signature algorithm and the signature itself.

The Public key infrastructure represents one of the most widespread uses of such certificates, namely X.509. In a PKI certificates are used to prove whether or not a certain website is legitimate or not. If the website is a legitimate one it will have a Certificate that was signed (usually using RSA) by a Certificate authority. The client will ask the CA for the respective public key and then verify it. If it is valid, then a connection will begin, if not, then the websites certificate either expired or it was either removed due to certain reasons, or it was forged. It is also possible to lose the ability to request the CA to sign a certain certificate.

2.8 Symmetric Cryptography

Symmetric cryptography as opposed to asymmetric cryptography follows one simple rule: the key that was used for encryption should be the only key that can be used for decryption. Keys that are considered secure also tend to be much smaller (256-bit for AES vs at least 2048-bit for RSA). All this at the cost of increased complexity of key management. Because of the way the whole process works the keys must either be pre-shared, or it must be encrypted with another key before it is sent over the channel to assure the confidentiality. Any breach is also disastrous as there would be no way to directly tell if the key has been compromised or not.

Some popular symmetric-encryption algorithms include: AES, DES, 3DES, RC6.

There are a few other advantages of symmetric ciphers [6]:

- Strong keys are cheaper to produce.
- Smaller keys.
- Quick encryption/decryption (Especially on hardware implementations, AES for example can complete the encryption of a block in a single clock while also requiring little memory).

2.9 Block cipher: modes of operation

Electronic Code Book mode (ECB). The plaintext is split into multiple blocks and after that they are encrypted individually using a block cypher (AES for example) (the block size depends on the algorithm used to encrypt it). Order of encryption is irrelevant as long as the resulting blocks are combined at the end. The decryption process consisting of the reverse of the encryption operations operations. This mode by itself is vulnerable to substitution attacks (the attacker can change the ciphertext without knowing the key to deceive the receiver).[7]

Cipher Block Chaining mode (CBC) adds in a certain level of randomness in order to avoid having the same ciphertext for repeating inputs. The randomness here being the Initialization Vector (IV) which does not need to be a secret, but it should only be used once. Using it multiple times will defeat its purpose as the same input will have the same output if encrypted more than once. IV can be generated in multiple

ways. XOR is applied over the first block and IV, the resulting block is then encrypted using the block cipher. For the rest of the blocks the result is used again as an input. The previous result is now used the same way IV was for the first block (new-block XOR encrypted-old-block and then encrypted). The decryption process is the opposite of these operations.[7]

Output Feedback mode (OFB) attempts to transform block ciphers into stream ciphers. First, the block cipher is applied over the Initialization Vector then the result is XORed with the with the plaintext resulting in the stream cipher (the plaintext must be split in n-bit blocks according to the block cipher used). The next key stream is obtained by using the previously generated blocks as an input to the block cipher. Both encryption and Decryption are the same.[7]

Cipher Feedback mode (CFB) is similar to OFB but instead of using the previous key stream the cipher is applied on the ciphertext again. Initially the IV is encrypted using the block cipher, after which the result is XORed with the plaintext block then the cipher text is fed back into the block cipher XORing the result with the next plaintext block. This process is repeated until there are no more plaintext blocks. Just like OFB encryption and decryption are the same.[7]

Counter mode (CTR) was introduced by Diffie Hellman in 1979 it is another way of turning a block cipher into a stream cipher, therefore, it is similar to the OFB and CFB modes. It introduced another value named Counter value which is combined with the IV, encrypted using the block cipher and then the plaintext block is XORed with the result.[7]

2.10 Advanced encryption standard

AES (Advanced Encryption Standard) is a block cipher which was proposed by Joan Daemen and Vincent Rijmen in 1999. It originally had the name of Rijndael and it supports multiple key sizes (128-bit 192-bit and 256-bit), in its initial form the block size could also be increased indefinitely to any multiple of 32-bit. It was also specifically designed to be easy to understand and to be easy to implement on both hardware and software.[8]

AES has a simple structure, and it works by applying a multitude of subroutines on the block until the ciphertext is obtained:

1. KeyExpansion – a subroutine used for generating/expanding more keys using the initial one so that every round can use a different key.
2. AddRoundKey – a simple XOR between the key and the block.
3. SubBytes – a substitution where each value is replaced with another one.
4. Shiftrows – a transposition.
5. MixColumns – a mixing operation.
6. Addroundkey.

Depending on the key size the algorithm will employ a certain number of “rounds”. Each round consists of repeating the steps 3,4,5,6 until the final round is reached. At that point only 4,5,6 will be applied.

The number 10 was chosen because, at the time, anything less than 6 rounds would negatively impact the ciphers security, and the 4 rounds were added on top as a security measure. Daemen et al. said that the diffusion of data can be achieved in just 2 such rounds, and 3 rounds for blocks of larger sizes [8].

Preparation steps:

Consider the 128-bit key be the following string: 0123456789012345

The content is placed into a table and then transformed into hexadecimal:

0	4	8	2
1	5	9	3
2	6	0	4
3	7	1	5

0x00	0x04	0x08	0x02
0x01	0x05	0x09	0x03
0x02	0x06	0x00	0x04
0x03	0x07	0x01	0x05

The exact same operation is applied over the plaintext.

The AddRoundKey step:

XOR is applied between the key and the current block as follows:

$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$
$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$
$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$

XOR

$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$b_{0,3}$
$b_{1,0}$	$b_{1,1}$	$b_{1,2}$	$b_{1,3}$
$b_{2,0}$	$b_{2,1}$	$b_{2,2}$	$b_{2,3}$
$b_{3,0}$	$b_{3,1}$	$b_{3,2}$	$b_{3,3}$

=

$k_{0,0} \otimes b_{0,0}$	$k_{0,1} \otimes b_{0,1}$	$k_{0,2} \otimes b_{0,2}$	$k_{0,3} \otimes b_{0,3}$
$k_{1,0} \otimes b_{1,0}$	$k_{1,1} \otimes b_{1,1}$	$k_{1,2} \otimes b_{1,2}$	$k_{1,3} \otimes b_{1,3}$
$k_{2,0} \otimes b_{2,0}$	$k_{2,1} \otimes b_{2,1}$	$k_{2,2} \otimes b_{2,2}$	$k_{2,3} \otimes b_{2,3}$
$k_{3,0} \otimes b_{3,0}$	$k_{3,1} \otimes b_{3,1}$	$k_{3,2} \otimes b_{3,2}$	$k_{3,3} \otimes b_{3,3}$

The SubBytes step:

A substitution is applied over the resulted block using a substitution box which is derived from the multiplicative inverse of $GF(2^8)$ combined with an invertible affine transformation and must avoid any fixed points($S(a_{i,j}) \neq a_{i,j}$).

The Shiftrows step:

The rows of the block are shifted to the right with different offsets. Row 0 is not shifted, Row 1 is shifted to the right c1 byes, row 2 by c2 bytes and row 3 by c3 bytes. The offsets c1, c2, c3 depend on the block length. For a 128-bit block length c1=1, c2=2, c3=3.[8]

The MixColumns step:

Here the columns of the block are considered polynomials over $GF(2^8)$ and the operation can be written as the following matrix multiplication:[8]

$$\begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} * \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

After all the final round is completed the ciphertext is obtained and can be decrypted by applying all the previous operations in their reverse order.

2.11 Zero knowledge

A protocol consisting of 2 parties, A and B, where A is the "prover" and B is the "verifier", is considered zero-knowledge if A can prove to B that he knows a certain secret without revealing any information about it.[9] In practice this can be done by having both A and B keep some sort of mathematically linked secret. B can find out if A holds the secret by issuing a challenge.

Goldwasser, Micali and Rackoff said that a zero-knowledge protocol must have the following properties:

- If the prover generates the correct result, the verifier will accept it.
- The verifier will only accept the result if it is correct.
- The verifier must not learn anything about the secret.

In the context of cloud storage, zero-knowledge encryption means that no one has access to the keys except the user, and nobody can learn anything about the keys.[10]

2.12 Lazy re-encryption

Lazy re-encryption (or delayed re-encryption), as described in [11], consists in delaying the expensive re-encryption of data as much as possible to reduce the possible waiting time generated when a user is removed from a certain file sharing group. The files will be re-encrypted whenever a change is done to them. This method allows the removed user to continue to access the files that were not changed leading to possible problems, especially if the group owners are not aware of this mechanism. In the case of a key becoming compromised Cepheus [11] forces an immediate re-encryption.[11]

For example, a certain user is added by mistake to a private group sharing sensitive data who is then immediately removed. If the group administrator is aware of this mechanism, he can proceed to modify every file in the collection to force a re-encryption. Process which can be very tedious, especially if the number of files is large enough to cause a slowdown in case of an immediate re-encryption. If the administrator is not aware and the removed user attempts to access the files, he will then be able to view

the sensitive content. Depending on the way the process is implemented, he may even be able to destroy data.[11][12]

2.13 Nested MAC construction

NMAC is defined by M. Bellare et al. as a function that works on an input of any lengths as follows [13]:

$$NMAC_k(x) = F_{k_1}(F_{k_2}(x))$$

Where:

$$f_k \text{ is a keyed compression function } (\text{takes an input of length } n \text{ and outputs length } l)$$

$$F_k \text{ is a keyed hash function whose input is split into blocks,}$$

$$\text{and each block is hashed by the keyed compression function}$$

When NMAC was initially described in [13], it was thought to be secure even when MD5 and MD4 were used. However, as time went on, better attacks have been discovered and both algorithms are no longer considered secure. It is also possible to use one algorithm for the internal function and a totally different one for the external function. One downside of this construction is the need to modify the code for the hash function due to the way that f and F work (f is applied on the blocks before they are fed into the hash function). In the paper, NMAC has served as a base for HMAC.[13]

2.14 Digital Signature Algorithm

DSA is often compared with RSA as NIST adopted it as a Digital Signature standard in 1994 instead of it. DSA is based on the discrete logarithm problem, while RSA on prime factorization.[15]

As opposed to RSA, the DSA algorithm does not provide encryption, only signing. The patent now belongs to the USA and it is available worldwide, royalty free.

The algorithm, as described in [14] has 3 parts:

1. <u>The Signing algorithm:</u>
    1. *Generate secret value k (0<k<q)*
    2. *Choose p such that p-1 is a multiple of q (p does not need to be a secret)*
    3. $generate\ h\ such\ that\ h^{(p-1)/q} \neq 1\ mod\ p$
    4. $g = h^{(p-1)/q}$
    5. $r = \left(g^k mod\ p\right) mod\ q$
    6. *Apply H(m) (H has an output of 160 bits or less)*
    7. $s = k^{-1}(H(m) + xr) mod\ q$
    8. *Generate x (0<x<q) (x is the private key)*
    9. $y = g^x\ mod\ p$ *(y is the private key)*
    10. *Send m,r,s (where (r,s) is the signature)*
2. <u>The Signature verification:</u>

1. *Receive m,r,s*
2. *If s OR* $r = 0 \ mod \ q$ *then reject, otherwise continue (bitwise or)*
3. $u_1 = (H(M))(s)^{-1} \ mod \ q$
4. $u_2 = (r)(s)^{-1} \ mod \ q$
5. $v = (g^{u_1} y^{u_2} \ mod \ p) mod \ q$
6. $w = v \ mod \ q$
7. $if \ r = w$ *then the signature has been successfully verified, otherwise reject.*

3. A hash function similar to SHA-1

2.15 Cocks IBE scheme

Identity-based encryption is like asymmetric key encryption in a sense that the user has 2 keys: a public key and a private key. In identity-based encryption the public key is the user's identity which can be a name, an email address, or any string which can uniquely identify an entity. This advantage comes with certain downsides such as requiring a centralized key generator, which in turn has access to all the public/private key pairs.

The Cocks IBE scheme bases its security on the quadratic residue problem. It works as follows:[30]

- First the entity that handles key generation/distribution generates 2 secret primes $p, q \equiv 3 \ mod \ 4$ and then calculates the public modulus n = p*q. After that, a public hash function is selected.
- Participates will then send their identity towards the above-mentioned entity, which applies the hash function multiple times to generate a value *a* such that $\left(\frac{a}{n}\right) = +1$. Next another value *r* is calculated as: $r = a^{\frac{n+5-(p+q)}{8}} \ mod \ n$. This value must satisfy one of the following equations:
  - $r^2 = a \ mod \ n$
  - $r^2 = -a \ mod \ n$
- Every participant receives their respective *r*.
- The scheme encrypts the plaintext in a bit-by-bit manner. So, for the current bit *x* where $x \in \{-1,1\}$ (0 will be coded as -1), 2 random values, t_1 and t_2, will be generated such that: $t_1 \neq t_2$ and $x = \frac{t_1}{n}$ and $x = \left(\frac{t_1}{n}\right)$ and $x = \left(\frac{t_2}{n}\right)$. The resulting encryption of one bit *c* is a tuple of 2 values c_1 and c_2 where:
  - $c_1 = (t_1 + \frac{a}{t_1}) \ mod \ n$
  - $c_2 = (t_2 - \frac{a}{t_2}) \ mod \ n$
- The decryption process is dependent on whether $r^2 = -a \ mod \ n$ or $r^2 = a \ mod \ n$:
  - If $r^2 = a \ mod \ n$ then $x = \frac{c_1 + 2r}{n}$
  - If $r^2 = -a \ mod \ n$ then $x = \frac{c_2 + 2r}{n}$

2.16 Fiat Shamir identification scheme

This identification scheme considers 2 parties: the prover and the verifier. The prover chooses a prime number *n* and then generates *k* random secret numbers (s_1 ... $s_k \in Z_n^*$), calculates the public vector *v* ($v_i = \frac{1}{s_i^2} \ mod \ n$) and sends it to the verifier.[31]

The verification protocol works as follows: [31]

1. The prover generates a random number *r* calculates $x = r^2 \ mod \ n$ and sends *x* to the verifier.
2. The verifier generates a series of k bits (e_1 ... e_2) and sends them to the prover.
3. The prover calculates $y = r * \prod_j s_j^{e_j}$ and sends it to the verifier.
4. The verifier identifies the prover successfully only if $x = y^2 \prod_j v_j^{e_j}$.

2.17 NIST Cloud Computing Reference Architecture

Cloud computing services can be complex and confusing, usually containing a multi-level architecture. The NIST Cloud Computing Reference Architecture attempts to organize the underlying components into multiple categories to facilitate understanding.

Firstly, there are 3 main categories of services such a system can provide [24]:

- Cloud infrastructure as a service (storage, backup, computing)
- Cloud software as a service (email applications, billing applications, document management)
- Cloud platform as a service (databases, integration)

Next, some general properties of efficient and accessible Cloud Computing services [25]:

- On-demand self-service. A user should be able to benefit or/and use the capabilities provided by a Cloud Computing Service at all times, without the need of manual interaction from the provider.
- Broad network access. The ability to access the service with a large spectrum of devices.
- Resource pooling. Clients will be served from a total pool of resources depending on their needs. They will not get to choose where in the system are those resources located.
- Rapid elasticity. The ability to respond and satisfy the client's needs as they arise. For example, in a Cloud Storage system, this represents the ability to easily increase the total storage in order to meet the customer's demand.
- Measured service. The ability to monitor and measure the performance of the service adapting it accordingly.

The authors of [24] split the cloud architecture into 5 actors: the consumer, the provider, the auditor, the broker, and the carrier.

The consumer, found at the foundation of such systems, represents an individual or a group of individuals (such as companies) that choose what cloud provider to use, what service they need, and use the service chosen. Most of the time, consumers also must pay a certain sum at a certain interval in order

to continue using the cloud service. They also have the possibility of changing the provider if a better offer appears somewhere else.[24]

The provider, as the name implies, manages, and delivers the services to the consumer. In a Cloud storage system, this refers to the management of servers, hardware (including purchasing more if the need arises), the hosting infrastructure and running the required software to make the resources accessible to the consumer.[24]

The auditor performs an independent evaluation on the cloud service to verify if it complies with certain standards. The evaluation can either target a specific field (privacy, performance) or multiple fields.[24]

The broker is an intermediary between the consumer and the service provider in charge of delivering the services and performance optimization. Using a cloud broker instead of directly communicating with the cloud provider, can increase efficiency as it opens the opportunity for better negotiations. [24]

The carrier is just the transport layer or the network layer.

In real-world scenarios, the actors can combine together or split up even more (e.g. the server management can be done by a third party instead of the provider), and one entity can play multiple actors.

There are also multiple ways to deploy a cloud in [24]:

- Public cloud. Servers all clients.
- Private cloud. Only serves a specific group (e.g., paying customers).
- Community cloud. The users also contribute to the overall hardware, and only the community members can access the cloud resources.
- Hybrid cloud.

Security and privacy are very important features that usually come at elevated pricing. So far, all cloud service providers fall into 2 extremes: either the provider has too much information and can easily abuse this to read the users files or it has too little information to do anything, and nobody can decrypt the information under any circumstances (excluding the brute force approach). This can be dangerous in case of terrorist groups using such a service to facilitate their operations.

2.18 ICE (Interactive Connectivity Establishment) protocol

The ICE protocol is an offer/answer protocol used in establishing media connections through NAT. There are a multitude of solutions that can serve this purpose; however, they are all dependent on the topology of the network. ICE is meant to be a fits-all solution. In the case of a device that lacks the necessary resource for the full implementation, a lite implementation is also defined [23].

ICE steps [23]:

1. Sending the initial offer

Before the offer can be sent, a user must first gather all the candidates (one candidate represents one transport address). Every candidate will also receive a unique priority and a base (the authors of [23]

also recommend a formula for computing the priority value). If 2 or more candidates with the same transport address and base exist, all the duplicates will be removed.[23]

Factors that can influence the priority of a user [23]:

- Whether the address is IPV4 or IPV6. (even though ICE can work with both IPV4 and IPV6, the latter receives an increased priority)
- Security

ICE uses SDP (Session Description Protocol) messages during the offer stage (steps 1, 2, and 3).

Example (taken from the standard) where *v* is the protocol version, *o* is the sender and a session id, *s* is the session name, *c* the connection type and other connection details, *t* is the time to live, *a* is a session attribute, *m* media details and *b* stands for bandwidth information [23]:

v=0

o=jdoe 2890844526 2890842807 IN IP4 10.0.1.1

s=

c=IN IP4 192.0.2.3

t=0 0

a=ice-pwd:asd88fgpdd777uzjYhagZg

a=ice-ufrag:8hhY

m=audio 45664 RTP/AVP 0

b=RS:0

b=RR:0

a=rtpmap:0 PCMU/8000

a=candidate:1 1 UDP 2130706431 10.0.1.1 8998 typ hot

a=candidate:2 1 UDP 1694498815 192.0.2.3 45664 typ srflx raddr 10.0.1.1 rport 8998

2. Receiving the initial offer

The receiver must now make sure ICE is supported by both agents. Then he must determine his own role, this can be either controlling or controlled. The controlling role generates the candidate pairs. A candidate pair is made up of 2 candidates, one which is sent by the other agent (remote candidate) and one of his own (local candidate). Such pairs must also be pruned. The controlled role just complies with the generated pairs. In the case where one participant uses the lite implementation, and the other uses the full implementation the former one will always take the controlling role and the later the controlled role. Afterwards the same operations performed by the sender are also made here (candidate gathering, prioritization).[23]

3. Receipt of the initial answer

When the answer returns to the offer sender, he will also make sure that ICE is supported by both candidates and the determine his own role.

4. Connectivity checks
5. Processing

After the previous steps are completed, the controlling agent can start nominating pairs. This can be done in 2 ways:

- Regular Nomination
- Aggressive Nomination

Using Regular Nomination an agent will evaluate the pairs, based on some metrics set by him, and select the most fitting one. Aggressive Nomination simply picks the first valid pair encountered. [23]

Once a pair is nominated, the media transfer can begin between the 2 candidates.

6. Media Transfer

3 Comparison

3.1 Tresorit

This section will begin with the description of some scientific articles that explain certain components that are used in Tresorit.

3.1.1 Tree-based Group Diffie-Hellman

Relying on one central authority (or a trusted third party) for key distribution comes with certain upsides (ease of use, avoiding additional computation/storage on the client side) and downsides (risk of attacks, especially DOS, the need for permanent availability, and depending on implementation, the central authority can store details about all the generated keys or even the keys themselves, which tremendously increases the damage done in case of a successful breach). TGDH is a contributory group key management where every group member must contribute an equal amount [20].

Cryptographic properties guaranteed by TGDH [20]:

4. Group Key Secrecy (the keys cannot be obtained by a passive attacker using brute force)
5. Forward Secrecy (a user that leaves the group must not know any of the new keys)
6. Backward Secrecy (a user that joins the group must not know the previous keys)
7. Key independence (if an attacker knows a subset of keys, he cannot find any additional keys using the information he has)

The communication channels are considered secure, and every message is signed.

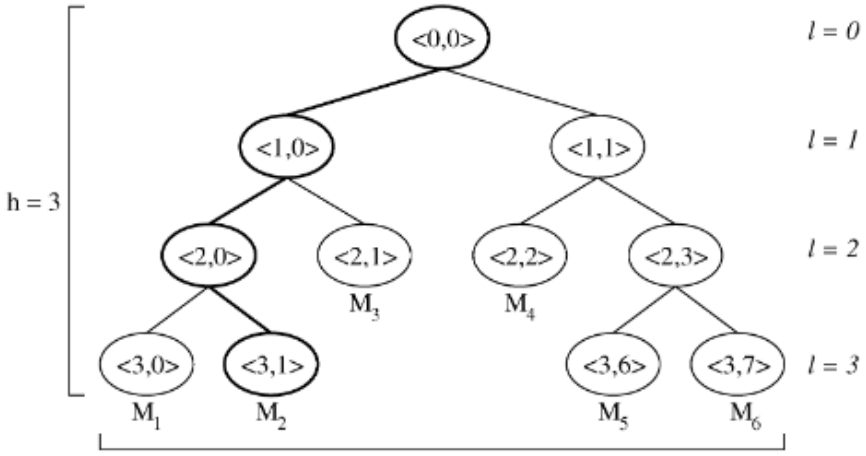

Fig 2.1. TGDH example [20]

Calculating the key at a certain non-leaf node is done using the key of one of the child nodes and the other child nodes public blinded key. Here the authors note that the root of the graph (the <0,0> node) should never be used as an encryption key. Instead, they recommend applying a strong hash function first, and then the result can be used as an encryption key. [20]

In TGDH there are 5 operations possible [20]:

- A member joins the group. (Join, Fig. 2.2)

1. A broadcast message is sent by the member who wishes to join the structure containing his respective public blinded key.
2. After the message is successfully received, the insertion leaf must be found. The chosen node must not increase the height of the tree (if not possible, the root will be the insertion node instead), being searched for from right to left. This node is also called a sponsor.
3. The sponsor recalculates the entire tree structure and all its blinded keys.
4. All other members update their own trees accordingly using the data received.

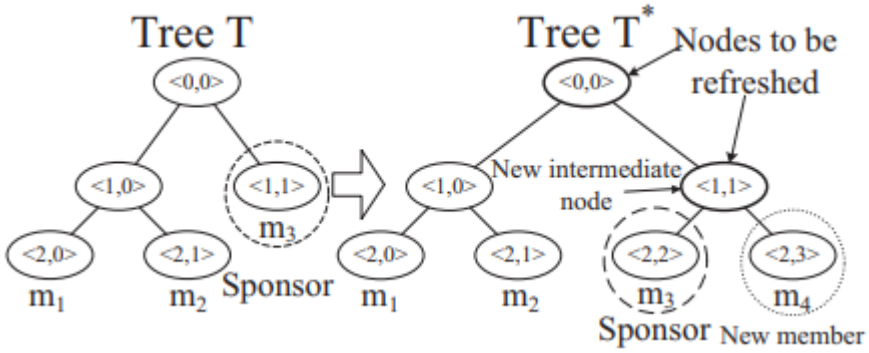

Fig 2.2. The join operation in a TGDH.[19]

- A member leaves the group. (Leave)
  1. This time the sponsor will be the searched for in the leaving nodes subtree whose root is located in the child node who shares the same root as the leaving node.
  2. All members will perform the deletion and update operation over their own tree.
  3. The remaining leaf node from the subtree where the deletion took place replaces the parent node.
  4. The sponsor recalculates the tree keys and broadcasts the new blind keys.
- The group is merged with another group. (Merge)
- The group splits. (Partition)
- Re-generating the group key (Key refresh)

The authors specify that the main purpose of the Merge and Partition operations is to deal with possible network problems and recovery.

	Communication		Computation		
	Rounds	Messages	Mod. Exps.	Signatures	Verifications
Join	2	3	3h-3	2	3
Leave	1	1	3h-3	1	1
Merge	$\lceil \log_2 k \rceil + 1$	2k	3h-3	$\lceil \log_2 k \rceil + 1$	$\lceil \log_2 k \rceil$
Partition		$\min(2p, \lceil \frac{\pi}{2} \rceil)$	3h-3		$\min(2p, \lceil \frac{\pi}{2} \rceil)$

Table 2.1. Communication and Computation costs in TGDH [20]

3.1.2 Invitation-oriented TGDH

TGDH in its unaltered form is not fully suitable for use in a cloud sharing environment because of the rigid way of choosing the sponsor [19]. When a user sends the request to join message there must be a way of identifying whether he has the permissions required to join the group. This can simply be solved by checking if there exists a member in the whole tree who invited the new user. However, this method can lead to delays as both the sponsor and the inviter may not be online at the same time, especially since the sponsor varies depending on the tree structure so the sponsor availability can also vary heavily. It would be ideal for a user to be able join the group as soon as he is invited.

The authors of [19] propose a modification more suitable for cloud sharing environments named Invitation-oriented TGDH which, like the original version, remains efficient and maintains the same cryptographic properties.

The authors introduce the notion of shadow node. Such nodes can only be leaves, containing a temporary private key to aid in the tree reconstruction without needing to use a specific node as a sponsor node. For example, in Fig 2.2, in order for m3 to add m4, m3 needs to recalculate the entire tree using their own private key and m4's blind key. This happens because m3 is the sponsor node. If m3 was not the sponsor node, the recalculation either cannot not take place or the sponsor node would need to communicate with m3 for them to recalculate the tree or share their own private key. However, because m3's private key should not be shared, the calculation cannot be done by the other sponsor node, so the new sponsor node has to wait for m3's response. If the sponsor uses a shadow node, they can now rebuild the tree without waiting for m3 (see Fig 2.3).

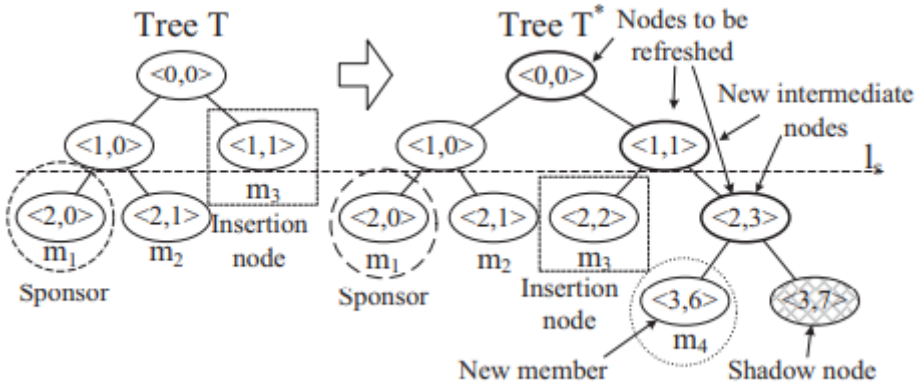

Fig 2.3. Node insertion using shadow nodes [19]

Tree balancing is done using red-white-black key-trees to help keep the computational complexity low.[19]

The join operation remains mostly the same. The insertion point will be the shallowest shadow node instead of the rightmost node that does not increase the height of the tree. The leave operation was changed to accommodate tree rebalancing.[19]

3.1.3 Secure-TGDH

S-TGDH is another modified version of TGDH that is adapted for group management in ad hoc networks. On top of some optimizations, the authors also added an authentication protocol.[22]

The authentication algorithm is an adapted version of another algorithm named Asokan Ginzboorg protocol (see [22] for more details) and is as follows [22]:

1. $M_n \rightarrow M_i: \{public_n, \mathrm{c1}i\}_p, n1, signature_n (i = 1 \dots n-1)$
2. $M_i \rightarrow M_n: \{public_i, \mathrm{c1}i, c2i\}_{public_n}, n2, signature_i (i = 1 \dots n-1)$
3. $M_n \rightarrow M_i: \{c2i\}_{public_i}, n3, signature_i (i = 1 \dots n-1)$
4. $M_i \rightarrow M_n: agree, n3, signature_i (i = 1 \dots n-1)$

Where:

- n1, n2, n3 are nonces.
- c1i and c2i are challenges.
- $\{x\}_P$ is the symmetric encryption of x with the key p.
- p is a weak secret shared by all of the users part of the TGDH tree.

First, an initiator node will be chosen (M_n in this case) in order to avoid the situation where every node sends a message to every other node which can lead to slowdown and other problems.[22]

In the first step, M_n encrypts his own public key and a challenge with p, alongside a nonce (used as replay attack protection), and a signature. When M_i receives the message, he will be able to verify the signature and decrypt the challenge. He will then generate his own challenge and encrypt it with M_n's public key together with his own public key and the initial challenge, increase the nonce, sign the message, and send it back. If the decryption was done correctly and M_i knew the secret p, then the decryption result should be the same as the challenge generated by M_n. This is how M_n checks whether M_i is part of the TGDH tree or not.

Something very similar happens at step 3, except now the public key sent by M_i is used instead of the secret p. If this check is also successfully passed, then every member will send back the *agree* message. The authors of [22] mention that step 4 can be optional, depending on the use of either TCP or UDP.

This method of authentication forces the group to be created based on a common password (the weak secret p). All the initial properties of TGDH are still present in this version.[22]

3.1.4 P2P group data sharing in Tresorit

Tresorit uses a File Level Module (FLM) that handles read permission and write permissions. Read permissions are offered in the form of a key to decrypt the files that were encrypted using AES with 256-bit key size using the CBC mode. Such keys are also generated by the FLM module with the help of a structure called hierarchical Key Lock Box architecture in order to avoid changing the master key when a specific file key is changed. Write permissions are verified using a digital signature provided by RSA. However, both of those algorithms can be replaced by something else. Lazy re-encryption is also used.

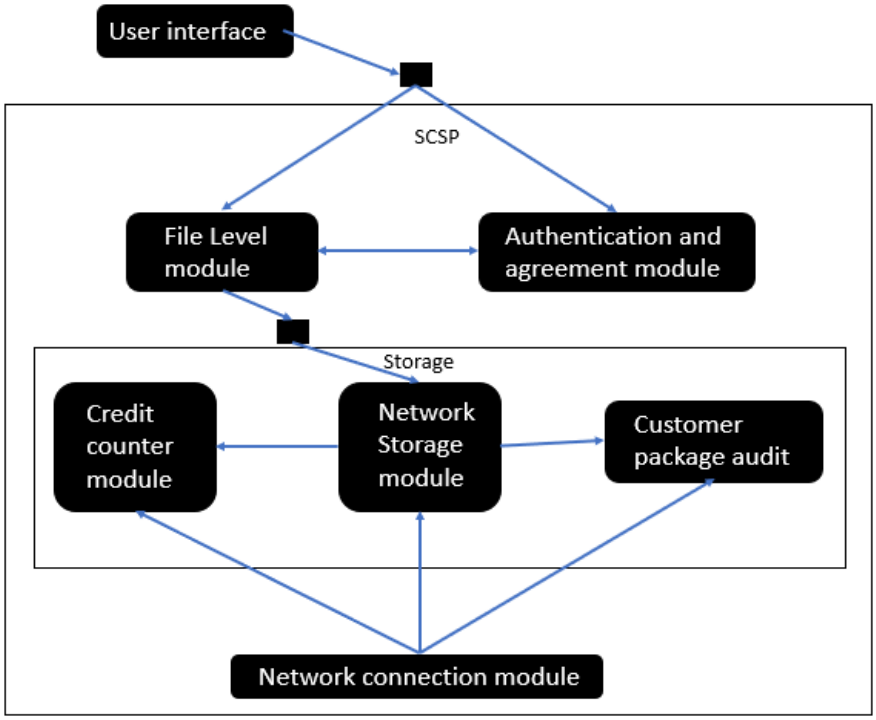

Fig 2.4. The module architecture. [15]

The communication is done over an authenticated channel, or on a channel built on an authenticated key exchange protocol. Not all the members from a group must be online at the same time when a new member joins or leaves the group. There is a module named Authentication and Agreement module (AAM) in charge of authentication and handling group changes. For authentication it uses Secure-TGDH or RSA based authentication.[16]

In order for a member of a group to access the contents of a file shared with that certain group, he must first specify the ID of said group. Then, an authentication procedure specified by the group will begin. Here, with the help of an auxiliary module, the user will send the authentication data which can consist of a password, an RSA private key or something else. This data will then be verified by the AAM module, and if it is valid, it will either return the Read Master Secret, the Write Master Secret or both. The secrets will then be used to decrypt file-specific keys or generate signatures. The FLM will also store the file-specific keys and signatures.[16]

The storage itself is divided into 3 submodules:

1. The Network Storage Module (NSM). It handles integrity, storage, searching and reading at a low level.
2. The Credit Counter Module (CCM) controls resource allocation. It prioritizes users who contribute more resources. Those who do not contribute anything may end up never receiving resources.
3. The Customer Package Audit (CPA) filters packages of paying customers so that they can receive enough resources even though they do not contribute anything.

There are also 2 types of Key Lock Boxes:

1. Read\Write Key Lock Box (which can be opened by a Master Read/Write Key)
2. Master Read/Write Key Lock Box (which can be opened using the Master Secret)

The CCM module tries to reward users who provide as many resources as they used, this is weighted using a variable named “gallant factor”. The ones who use many resources and provide little to nothing will be penalized by having their share revoked. The gallant factor is calculated using the opinions of other users. However, these opinions will also be analyzed and compared to the previous information given. If the information previously given was proven to be true, then in the future this user will be considered reliable. Unknown peers have the reliability factor set by default to medium low.

Note that due to the system being P2P all the modules will reside on the client side, as the existence of a main server is not mentioned. Tresorit also uses this scheme with some modifications. For example, there would be only paying customers as it does not serve non-paying customers permanently. Users also do not need to share any resources of their own as the cloud offers all of them which means that at least the CCM and CPA modules would need to be modified. The location and implementation of the AAM module is also very important because it handles keys. If this component is located on the server-side, it would be very easy for the provider to obtain reading privileges. If it is on the client side, it is also very easy to leave a backdoor that sends the keys to the server whenever requested. Disproving the latter would require package analysis or decompiling the client.

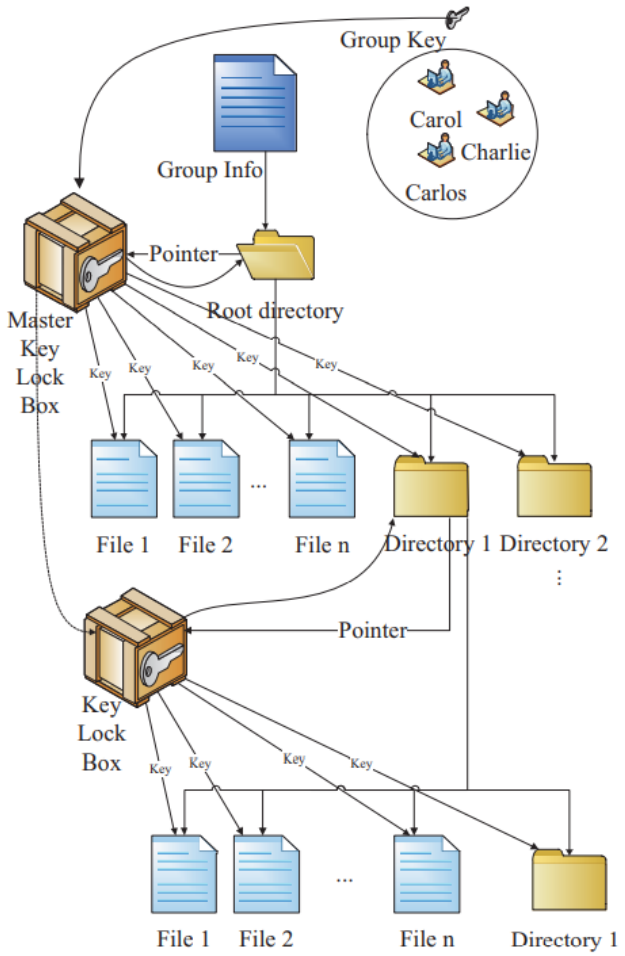

Fig 2.5. The Lockbox Tree Structure.[21]

The concept of Lock Boxes is used which can be split and arranged in a B-tree-like structure (derivate from a TGDH-type key) in order to maximize the access speed, or as an array where each key is associated to a file ID. Keys that are used more often will reside on the higher levels. Every Lock Box contains keys corresponding to certain directories. In case of a key becoming compromised all its child key nodes will be disabled and replaced.

As mentioned above Tresorit uses lazy re-encryption which has one main flaw, the ability for a removed user to change/delete files/file contents until the key is changed for that specific file. Here the authors considered that if a situation similar to this occurs, immediately changing all the write keys would still give the removed user a fairly large window where he can perform such actions (the authors considered that the attacker has a considerable speed advantage over the scheme). Therefore, it is highly recommended by the authors to only grant write permission to trusted members.[16]

In practice, an attacker would only hold the speed advantage if he were ready to react as soon as he was removed from the group, which is unfeasible for the attacker unless he has a very specific goal in mind. However, a record should remain regarding which user has modified a specific file or who was the user that has uploaded the file together with a time stamp. So, if an attacker is kicked from the group and he makes a modification immediately after, the legitimate group members will be able to check what files have been modified and when were they modified. Faking such records would require knowledge of another user's key. The most dangerous scenario here would be the insertion of malware that runs automatically when it is downloaded. But the malevolent user has no need to be kicked from the group as he can easily do this while still part of it. On the other hand, immediately switching all the write keys at once can lead to potential DOS, if the tree-structure is large, depending on which user performs the computations. In conclusion this solution would prove useful against extreme edge-case scenarios bringing with it other, more serious security threats, hence why it is not used.

In order for a user to access the keys, he will first send his personal key to the AAM module. If the key is valid AAM will return the Master Secret, if it is not, it will return an unusable Master Secret. With it he can then open the Master Lock Boxes which contain another key which will open another Lock Box containing the decryption key for the root directory. In case of a group setting, this box will also contain a special key to a merged group, which any other member can calculate using their AAM even if keys belonging to other members change. (For more details see [16])

3.1.5 Tresorium

Tresorium is the cryptographic storage system used by Tresorit.

Most cloud vendors maintain some sort of control over the users and the data they store on the servers (e.g.: the ability to view the contents of a stored file, the ability to modify ACL lists) which usually implies violating a user's privacy. The severity of this violation depends on a case-by-case basis. While most cloud vendors retain the ability to view the data stored on the servers only to make sure it is conformation with the EULA, others choose to use it for targeted advertising, and in rare cases they even retain the ability to share the data.

The same Lock Box key structure detailed above also appears here. Its structure is going to be the same as the directory structure it holds keys for. The Master Lock Box (this box can only be opened by the key shared by the group) will contain keys to decrypt the contents of the root directory, alongside the keys to open the Lock Boxes that reside under it in the tree structure. The group key is calculated using Invitation-oriented TGDH.

If the Authentication and Agreement Module only resides on the client's machines, then the cloud can have access to the keys, and therefore has access to the decrypted data. The reason for this is because the AAM returns the key for a Lock Box which in turn holds the key for decryption.

The paper presents 3 algorithms with different purposes [21]:

1. Algorithm 1 is used to determine the dirtiness of a key. (For the exact structure see 21)

A key being marked as dirty means that it must be replaced the next time someone makes a modification to the file. Due to the way Tresorit handles files (the user can encrypt/decrypt files only on his own machine) using lazy re-encryption solves 2 problems at once. The secrecy is maintained because only the user will be able to change the keys, and the system is not overloaded because the keys are replaced only when a change is made to the files contents. Because of this mechanism, however, write permissions should be granted with great caution as a bitter user can modify all the files he has access to if they were not re-encrypted, despite not being part of the ACL (access control list) anymore.

2. Algorithm 2 handles group changes.

Here the user that invited/removed another user generates a new key which is then used to encrypt the master key lock box.

3. Algorithm 3 which is related to the lazy re-encryption.

This scheme has the following properties [21]:

- Forward Secrecy
- Weak Backward secrecy (useful when a new user is added to the group)

The scheme remains fairly efficient, as the upload time in the environment used by the authors favors Tresorit in comparison to Google GDrive and Dropbox.[21]

3.1.6 Tresorit workflow

First, at registration Tresorit generates 160 random bits, called cryptographic random salt. This is done in order for the information stored in the database to contain as little details as possible about the actual password.

The password is combined with the random bits after which the PBKDF2-HMAC-SHA-1 is applied over the input, then the function is ran over the result again. This process is repeated 10,000 times on the user's client. The final result and the 160 random bit string are then sent to the database for storage in order to be used in the Zero-Knowledge authentication process which works as follows:[17]

1. The user notifies the server that he wishes to log in.

2. The server sends a challenge request and the 160 random bit string that was used during the registration process.
3. The user applies the PBKDF2 function the same way he did when registering and then sends the result to the server.
4. The server checks if the received value is equal to the one stored in the database. If yes, the user is successfully logged in.

After the user is logged in, a secure TLS channel must be established before data exchange can begin. Tresorit uses a version of TLS (named Hardened TLS) that allows for TLS authentication for the users as well, not just the servers. The authors mention that this is done in order to deny attackers the ability to attempt an impersonation and try to access the files on the cloud, even though they would not be able to decrypt the contents because of the End-to-End encryption provided.[17][18]

In Tresorit only the user decrypts and encrypts data, and because of this, only the client knows the keys required for such operations and also has the ability to choose what encryption key to use for a certain file. The cryptosystem used is AES-256 in CFB mode. As mentioned above, this mode of operation requires an IV (initialization vector) which, if reused can lead to information leaks. This is avoided by using a new IV every time something is encrypted.[17][18]

Before the encrypted file is sent to the cloud, a MAC (HMAC-SHA512 in this case) is generated for the file and then concatenated with the message to ensure integrity. Afterwards the cloud receives the files and inserts to the storage while maintaining their structure intact.

The architecture described in the **P2P group data sharing in Tresorit** section is used here, so first the Lock Box tree structure must be updated. If the Authentication and Agreement (AAM) is based on RSA the module will also contain a pre-master secret (or multiple secrets if the file is shared with a group) and the user-specific certificate/certificates. If based on TGDH it will only store the Diffie-Hellman user certificates. [18]

The key recovery process is similar as above:

1. The AAM receives some data to decrypt the Master secret.
2. The Master secret will then be used to decrypt the file/directory specific keys to access their contents.

For efficiency reasons the actual stored files are encrypted using AES256, and in order to avoid complications related to key management, the Lock Boxes use asymmetric cryptography.

A user can invite another user to his group by sending an email after which a 3-step authentication process will begin [18]:

1. Users will authenticate each other using the RSA key and an X.509 certificate belonging to each of them. This is also where a long-term symmetric key is established which can be reused for other invitations. The inviting user also sends a 256-bit secret.
2. The invited user now must prove that he received an invitation by responding to a challenge using the secret received in step 1. If the answer is accepted, then he is added to the group.

3. (Optional) The users can choose to use a pre-shared password for additional security.

A user can also be removed from the group and have his abilities to read and modify files nullified. In a normal environment this can happen for a multitude of reasons, some more benign than others. In Tresorit this procedure also has 3 steps [18]:

1. The user is removed from the groups access control list. This step is present in almost all cryptographic cloud storage services that have some form of group management/sharing. Normally, this step would be sufficient if the users put their trust into the cloud service.
2. The removed user's certificate is deleted from the AAM, and the user who performed the removal changes the keys in the Master Lock Box and re-encrypts it. This way the removed user does not have access to the new root directory anymore. But, using the cache memory, he can still view certain files that were accessed recently (if their contents were saved in the cache).
3. The dirty flag is applied to all the files the removed user had access to so they can be slowly re-encrypted with a new key as they are modified by a user with write privileges.

3.2 Google Cloud

Google Cloud is one of the biggest and most popular professional cloud service providers who not only offer storage, but also Platform as a Service, and Software as a Service. The security is split into 4 layers: Infrastructure security, encryption at rest, encryption in transit, application layer security.

3.2.1 Infrastructure security and other aspects

On top of guaranteeing security of physical premises by restricting access, Google Cloud makes sure that all machines and the software on those machines has not been tampered with using cryptographic signatures. Every physical machine can be uniquely identified, and all the signatures are either rooted in a microcontroller or a Google-designed Security chip.

The service orchestration is made by a system called "Borg", which does not consider any service running on the infrastructure as trustworthy. All the services are cryptographically authenticated and authorized using a set of credentials before communications between them can take place.

Because Google Cloud provides other types of cloud services on top of cryptographic storage (such as Platform as a Service) a stronger degree of service separation and sandboxing is needed. Some techniques used by Google Cloud for resolving this issue are: Linux user separation, language and kernel-based sandboxing and hardware virtualization. The solutions, and the amount of separation chosen depends on the security threat of the task or the service that will be executed. Note that in a cryptographic cloud storage service hardware virtualization can be enough if the provider makes sure that execution of any stored file is not possible. In normal circumstances, in Tresorit this would not be possible anyway, as files are only encrypted/decrypted on the user's machine. A user can, however, send a plaintext file falsely labeled as encrypted which then runs automatically when stored. Access Management between services can be done using a construct very similar to access control lists. An identity is issued to every service and personnel, and depending on it they are either granted or denied access to a certain service. All inter-service messages are encrypted before they are sent. The level of protection is also customizable, ranging from only integrity

checks to full on double-layered encryption. Encryption is also done at the application layer to generate another degree of separation. Keys for this encryption are managed by a central key management service which supports automatic key rotations.

Only a small amount of the infrastructure is directly connected to the internet in order to allow implementation of advanced filtering mechanisms and explicit DOS protection [26] (which relies on package filtering and package throttling). This feature is not explicitly offered in Tresorit. High availability percentages are always very important in any Cloud service, but they are arguably more important in Platform as a Service or Software as a Service based clouds because such services are hard to replace on a very short notice. Whereas a Storage Service downtime can be temporarily resolved by storing data locally.

Authentication is somewhat rudimentary compared to Tresorit's zero-knowledge version. A user simply inserts his username and password. On top of that, Google keeps track of the sign-in devices and sign-in locations, which implies storing the location or the IP address of the sign-in location. There are a multitude of methods to verify whether a successful log-in took place on a device, some of them being: storing some data that can uniquely identify the terminal such as MAC address storing (which can be faked without great difficulty) or storing some data on the user's device (which can be considered intrusive). Both location storing and device fingerprinting compromise a user's privacy. Tresorit uses a Public Key Infrastructure to solve this issue without storing location data. There, every device will be authenticated using multiple x.509v3 certificates. It is not specified how the password is stored or what hashing algorithm is used to store it if the classical approach is chosen. Moreover, because authentication is not zero-knowledge, the provider already gains a lot of information about the users. Two-factor authentication is present in both services.

Because of the amount of data stored, great investments are made in order to keep it safe [26] such as mandatory use of phishing resistant One Time Pads for employees, aggressive monitoring of infrastructure administrators and two-party approvals. Combining machine intelligence with a set of rules, intrusions can be detected quickly. The threat response team also operates non-stop, adding another layer of reliability. The approach made by Tresorit to this problem is to store and rely as little as possible on the provider. In case of a data breach, the attacker would only gain encrypted or irrelevant information about the users (such as the salted hash or the encrypted data stored), the personal email address being the only useful information gathered. The most dangerous thing that can happen is in the case of a serious system breach where the attacker gains the ability to view and modify code in the application. At first glance, Google Cloud holds the advantage here because of their tamper-proof software and machines, as such a breach would be quickly identified. However, Tresorit uses the Microsoft Azure infrastructure.

Hardware virtualization is done by an improved version of KVM stack specifically build for this task.

3.2.2 Encryption at rest in Google Cloud

Google Cloud encrypts all customer data and metadata, mostly using AES256 or AES128 with the CBC, GCM or CTR modes of operation, alongside a proof of integrity. The Google Cloud data platform

contains 4 layers: the application layer, the platform layer, the infrastructure layer, and the hardware layer. Data is broken up into pieces (which can be at most a few GB [26]) encrypted on all layers except the application layer. Every piece will have a different identification tag and key, after which it will also be encrypted before being stored in different places. Collisions are also extremely improbable (1 in 2^128 for AES256 and 1 in 2^64 for AES128). If any change in the data occurs a new key is generated instead of reapplying the same one. Similar to lazy re-encryption a user cannot force the re-encryption of every file automatically. Every chunk also contains an Access Control List which dictates who can access it.

Tresorit uses the TGDH structure for key sharing, only AES256 and the directory tree structure for general storage. TGDH trades speed for secrecy, as only the tree members will be able to reconstruct the key. The data tree structure can be faster depending on how fast the data chunks can be recovered.

In the case of a key becoming compromised, Tresorit is at a disadvantage. In Tresorit, the impact also depends on the level the compromised key was located at in the directory tree as every child node will also be compromised. Therefore, finding out the Master Key leads to entire tree being lost. The recovery process can also take a long time due to lazy re-encryption. However, the most feasible way for a key to become compromised is through targeted malware, making key management a user's responsibility. In Google Drive, the ACL can act as an additional layer of protection in this case, with the downside that, depending on implementation, the provider gains increased privileges and requires additional trust from the user. This is because generally, access control lists can be more easily abused than other cryptographic solutions, especially when physical access to the hardware is possible. Permissions must be granted carefully just like Tresorit, as one user could modify the ACL list and deny the initial user access to the files.

Encryption on the hardware layer is either AES128, if the data is stored on a hard drive, and AES256 if it is stored on an SSD. Some data is also stored in a plaintext form, such as console logs, core dumps, debugging logs (such logs can also contain customer data and metadata) and some temporary files [26]. In the case of cryptographic storage this data does not really matter. Because there is no file execution in place, they cannot generate such logs by themselves. In the case of Software as a Service and Platform as a Service this can be a potential privacy issue as they would require manual review to remediate the cause of the problem.

There exists a special library maintained by Google, named BoringSSL (which, in combination with Keyczar, a cryptographic library, implements the AES used in Google Cloud). The decision to branch from the OpenSSL library was made because too many flaws were found in it. [26]

Google Cloud uses two types of keys: data encryption keys and key encryption keys which are all managed by Google's key management service [26]. Both keys are generated using BoringSSL. The data encryption keys are used to decrypt and encrypt a data chunk. In order to increase the speed of these operations, they are also stored near the data chunk they were used on. The data encryption keys are then encrypted again by the key encryption key. The key encryption keys can be generated by either the management service or the by a storage service. They are stored in key management service (where they

can no longer be exported), their use is regulated by an ACL, and extensive logs are kept regarding who used it and what key was unwrapped. They can also be reused sometimes to make key management easier and faster. The decryption process is exactly the opposite: the encrypted data encryption key is sent to the management service which decrypts it if the entity who requested the decryption is on the ACL's, and sends the plaintext result back, after which the key is used to decrypt the data. All the keys are also change at certain time intervals. Google Cloud Storage renews the key encryption keys once every 90 days. [26]

Unlike Tresorit, the data encryption keys are not private. While the key transfer may take place on a local network, making it practically impossible to be intercepted from the outside (unless a security breach takes place), the Google Cloud has all the necessary information to decrypt and view the stored data: the encrypted data chunk location (or the ability the intercept it) and the decryption key. One easy way to avoid this, although not very efficient, would be to use asymmetric cryptography when the key is sent from the management service to the consumer in order to avoid it being easily intercepted or the consumer can encrypt the data locally before sending it for storage.

Backup files are encrypted with another data encryption key generated by the key management service and a random seed, this key is not related to the other key used to encrypt the actual data [26]. This way the data is always only visible in its encrypted form. In Tresorit, this is achieved by default as it would be impossible for the provider to decrypt the files in the first place.

All key encryption keys are also encrypted by another set of keys named "key management service master keys". They are AES256 keys (AES128 for older keys) and have their own dedicated storage named root key management service. Only 10-20 of them exist [26]. Those keys are protected by another key called a root key management service master key which is also AES256 and exists only in RAM. Steps required for decrypting a file:

1. Identify and gather all the encrypted file chunks from the memory space (for storage, chunks can have between 256 KB and 8 MB [26]).
2. Identify and gather all the file-specific encrypted data encryption keys (those are also the decryption keys).
3. Send the encrypted data encryption keys to the key management service.
4. The key management service then verifies if the user can perform the operation. If yes, the module then proceeds to identify all the required encrypted key encryption keys and then send them to the root key management service for decryption.
5. The root key management service performs the same check, sends the key forward to the root key management service master key distributor who then verifies if the key is being used properly, decrypts the key, and sends it back.
6. The root key management service then decrypts the key encryption keys and sends it towards the key management service.
7. The key management service decrypts the data encryption keys and sends them to the service.
8. The stored data is now pieced together and decrypted.

This entire process can be lengthy if it is repeated top to bottom for larger files. The first step being especially long as every chunk requires a unique key, whereas the second step can be shorter if only one key encryption key was used to encrypt all the data encryption keys. The time taken to decrypt the files in case they are shared would be very similar to the normal scenario. In Tresorit, the main thing slowing down the decryption is how deep is the specific file in the tree structure, as moving from level to level requires opening lockboxes and decrypting files. In case the file is shared it would depend on the number of users it was shared with. The one disadvantage of TGDH is that reconstructing the key can take more time if the file was shared with a large to very large pool of users.

3.2.3 Encryption in Transit in Google Cloud

Due to the sheer size of Google Cloud, and for additional protection, security measures used for handling local traffic differ from those used to handle outside traffic. To facilitate this approach, there is a specific module that only handles outside traffic, named the Google Front End. This module responds to HTTPS connections with TLS after the user certificate is validated. All data that is sent and received from Google Cloud services is authenticated (it is not specified whether authentication is mutual or one-way), encrypted, and contains an integrity check on top of TLS. This is somewhat similar to Tresorit, because of the Hardened TLS version used which provides mutual authentication. There, the data integrity is guaranteed by HMAC-SHA-512. In [26] it is also explicitly mentioned that Google uses a TLS implementation that provides forward secrecy. However, this is not necessarily an advantage, because, as of TLS 1.3 forward secrecy is built into the protocol. Moreover, as of March 2018 BoringSSL does not yet fully support TLS 1.3.[26] As of June 2017 Google also operates their root CA (central authority), which gives them the ability to sign certificates used in PKI's. The central authority first needs to be trusted by a large pool of operating systems and browsers for it to be usable. Because of this, Google still uses other third-party CA's.[26] In real case scenarios, CA's are only brought up online from time to time to reduce the chances of them becoming compromised. Therefore, it can be valuable for certain corporations to be able to sign such certificates whenever they want to, instead of having to wait for another CA to be brought online. Entities that successfully ran a TLS session in the past can choose to use a private session ticket or ID to connect without needing to repeat the whole process. Google routinely refreshes TLS certificates to further reduce to risk of an attacker hijacking a TLS session.

Data is encrypted at the network layer, using AES128-GCM, whenever it is sent from a virtual machine within the secure local network to a virtual machine outside the area where Google Cloud operates.[26] The encryption key varies from session to session and is established using ALTS. For authentication, a Security Token is used. This token is a combination of information that uniquely identifies the VM and a physical boundary secret (128-bit pseudorandom number). For communication between services, ALTS is used to encrypt, authenticate, and verify the integrity.

Google Cloud offers the user the possibility to configure the methods used for encryption in transit. The options range from using Google Cloud VPN to more in-depth TLS/SSL management. This feature is

not particularly important for data storage, as the user always has the ability to encrypt the data locally using a third-party software, and to also store the key locally. [26]

As for Post Quantum Cryptography, both storage services offer some protection against Quantum Computing. Both TLS and ALTS implement elliptic curve based key exchange algorithms, elliptic curve DSA for authentication and AES256, which are resistant to Quantum Computing, and support RSA-2048 authentication, which is weak against Quantum Computing. At a storage level however, Tresorit only uses AES-256 as opposed to Google Cloud having some data still encrypted using AES-128.

3.2.4 TLS 1.3 vs ALTS

As mentioned above ALTS is Google Cloud's TLS replacement for connections that take place within the physical infrastructure. The main reason behind its development in 2007, was the existing vulnerabilities inside TLS at that time. In TLS, a participant's identity is related to the hosts name, in ATLS everything has a unique ID which is embedded in a certificate. [26]

There are 2 types of certificates: a Master certificate and a Handshake certificate. Similar to a PKI, a Signing Service with a Master certificate will sign on request an identity certificate using the Master certificate's private key. The identity certificate will then be used to generate a Handshake certificate. The Master certificate generator is Google's CA mentioned above. The certificate lifetime depends on the entity: human Handshake certificates are issued by human users and last ~20 hours, Master certificates last a few months.[26] Permission verification takes place only during the Handshake process, where the entity that inspects the certificate also checks if the issuer is valid or not. Like the Hardened TLS variant, Certificate Revocation Lists are used to check for expired or deleted certificates, and it is possible to resume a session using a unique resumption key (session tickets in TLS).

ATLS contains 2 parts: the handshake and the encryption. The handshake has 5 steps [26]:

1. The client sends the ClientInit message to the server containing the handshake certificate and the supported protocols.
2. The server verifies the certificate, selects the protocols that will be used, computes the Diffie-Hellman key which will then be used to generate 3 session secrets (the authenticator secret, the resumption secret, and the key K used to encrypt and authenticate messages). Then, the server responds to the client with his own certificate.
3. The client will not immediately respond to this message, he will instead wait for the server to send the ServerFinished message consisting of a HMAC function applied over the authenticator secret and a string.
4. The client now calculates his own Diffie Hellman key. He should then be able to generate the exact same 3 session secrets and verify the HMAC function result.
5. The client applies the HMAC over the authenticator secret and a different string and sends it back to the server.

Both parties can now use K to encrypt and decrypt messages using AES-128-GCM/VCM. TLS supports only the GCM mode of operation for AES-128/256. Both protocols are, at least in terms of

workflow, similar. While TLS 1.3 does the key exchange, authentication and server parameters establishment in phases, ALTS merges all those phases into one. This lowers the number of messages sent (5 vs 7 steps for one-way authentication in TLS). Google explicitly mentions that ATLS is vulnerable to impersonation if an attacker finds out the Diffie-Hellman session key, giving him the ability to perform actions in the name of other entities.[26] The identity inside Handshake messages is also not private [26]. However, considering ALTS should only be used on a local network this is not necessarily a downside, as entities do not need to be anonymous. Moreover, this can help to easily identify peers in a history of messages, enabling easier debugging. The zero-roundtrip session resumption protocol was not included in ALTS as it provides weaker security properties in TLS 1.3.[26]

3.2.5 Conclusions

As a data storage service, Google Drive (which is part of Google Cloud) is widely used by other software such as Google Photos, and it is also a common data backup solution for phones running the android OS. This popularity makes it a very valuable target for attackers looking to extract certain data, (sometimes even personal data) therefore, requiring extensive protection against security breaches. On top of this, Google Cloud provides other types of services (Software, Platform). The security solutions used by them focus on protection against DOS attacks, abuse by employees with extensive logging practices, and minimizing the damage of potential data breaches by separating it and applying multiple levels of encryption and access control lists while also retaining the ability to view the customer data.

3.3 Microsoft Azure

Microsoft Azure is another popular Cloud service provider created by Microsoft. Just like Google Cloud it also supports Platform as a Service and Infrastructure as a Service. Tresorit also uses Microsoft Azure datacenters.

3.3.1 Infrastructure Security and other aspects

Similar to Google Cloud, the first security measures taken consist in the physical security of the datacenters. The data wiping process is also similar, anything that cannot be wiped is physically destroyed on the premises. Data is backed at 2 distinct locations and 3 copies of it are kept on every location in order to combat possible natural disasters.[27] The virtual machine management is done by a service named Fabric Controller. Verifying whether a VM has been compromised takes place during start-up using a Secure Boot procedure. This consists of checking all the components before loading [27] using digital signatures and 2 databases (one contains valid signatures, and one contains the revoked signatures). All of this is rooted into a chip named Cerberus, whose identity cannot be cloned. This essentially nullifies any option of cloning the chip, as opposed to Google Cloud's microcontroller/security chip where this is not specifically mentioned. Users also have the ability to encrypt VM hard drives which can be considered an additional line of defense in the case of a data breach, unless the key managing authority is also compromised.

Another similarity to Google Cloud lies in the way Microsoft Azure handles communication between services. X.509 self-signed certificates and TLS encryption are used to identify and communicate

between components.[27] Microsoft also has their own Central Authority, which unlike Google Cloud's CA, is backed by a trusted root CA. Meaning, the certificates can be used outside of the cloud's boundaries. The Fabric Controller obtains such a certificate and uses it to authenticate itself to other services, alongside a set of keys or passwords.

The network infrastructure contains 4 components: Edge network (similar to the Google Front end component in Google Cloud), wide area network (acts as a connection between servers in different regions), Regional gateway (acts as a connection between datacenters in the same region) and a Datacenter network. Access control is also supported at the network layer and consists of another firewall.[27] The reason for this design is the increased number of regions and datacenters (58 regions vs Google Clouds 24). Extensive DOS/DDOS protection is also present here. Multiple layers of firewalls are used to filter traffic and deny unauthorized access. The concept of forced tunneling is also used to further enhance the VM protection. Forced tunneling is used to deny VMs the ability to send requests outside of the local network while retaining the ability to respond to them.[27] This mechanism is used in order to avoid scenarios where the VM has been tricked into sending a request to a malicious user who then can respond by sending back malware.

Microsoft Azure runs a vulnerability scan roughly 4 times a year, on top of providing other monitoring software. Every build and component are also scanned for viruses using Microsoft Antimalware before they are deployed.[27] There is a program that awards users for finding vulnerabilities, remote code execution especially,[27] for a hardware virtualization tool named Azure hypervisor.

There are multiple options to protect data in transit such as using VPNs, TLS 1.2 and the later variant, IPsec, and others. However, depending on implementation, enabling TLS 1.2 can also enable downgrade attacks after which the attacker can then attempt to force weaker primitives available in TLS 1.2 leading to data potentially becoming compromised. Even without this, as mentioned above TLS 1.2 does not provide perfect forward secrecy. As of March 2021, Microsoft,OneDrive still uses TLS 1.2 which makes it vulnerable.

3.3.2 Encryption at rest in Microsoft Azure

Users can choose whether the encryption is done on the client-side, or the server-side. If the encryption is done on the client, key management becomes the user's responsibility. The user can also choose whatever encryption scheme is most suitable for the operations performed (choosing an unsecure encryption scheme can also facilitate attacks). Because, as mentioned above, Tresorit uses Microsoft Azure datacenters, it is highly likely this is the approach used by them. This way, the cloud does not have access to any of the keys either (assuming no implementation-level backdoors exist). Using server-side encryption, the client sacrifices privacy (and some transit security) for ease of use, as the server is in charge of key management and choosing and encryption scheme. When using server-side encryption the data is only encrypted by the TLS protocol by default, so if the TLS channel ever becomes compromised whatever data was being transmitted at the time can be viewed in plaintext by the attacker. Whereas with client encryption

the attacker would only be able to view the encrypted data. Tresorit's client-side encryption model provides both privacy and ease of use by default. Google Cloud only offers server-side encryption.

A mixed option that also suffers from the lack of security while in transit is available, where the server performs the encryption, and the user manages the keys. In this scenario if the key is lost the user also loses access to the data.[27] The key managing entity will be able to access the keys using an access control list, which results in the loss of the user's perfect privacy because the cloud provider can decrypt the data. The only benefit gained from this approach is the ability to refresh the keys whenever the customer wishes. In Tresorit the keys are refreshed whenever a group change takes place. However, the re-encryption will not be performed until the file that needs to be re-encrypted is modified. This can be a potential issue for organizations with large amounts of files stored that also do not get modified at all, as re-encryption may not take place in such cases.

In Azure Blob storage encryption at rest on the server-side is also done in a similar way to Google Cloud. The data is split into pieces, then every piece is encrypted with a unique data encryption key (AES-256), afterwards the data encryption key is encrypted with a key encryption key (can be symmetric or asymmetric, the exact type/algorithm is not specified) that never leaves a certain Key vault (this entity also in charge of managing VM encryption keys). Access to the key encryption keys is granted based on identity. The encrypted data encryption keys are also stored in the cloud. Users also have access to their data based on a secret private key. [27] This security model lies somewhere in between Google Cloud and Tresorit in terms of how hard it would be to acquire the plaintext data if a security breach happens. This is mostly due to the lower number of steps required to gain access to a key, as opposed to Google Cloud. Here, if an attacker does not hold any information about the data prior the breach he has to: identify the location of the data chunks (the exact chunk size is not specified), identify the location of the data encryption key chunks, acquire the respective key encryption keys, and then decrypt the data. Whereas in Google Cloud, on top of this the attacker must go through 2 additional key managing entities: the root key management service and the root key master key distributor. Because both services also use access control lists (role-based access control in Microsoft Azure) the easiest way to trick a key management entity into decrypting a key for an attacker, at least at first glance, would be to attempt to impersonate the identity of someone who is on the list, or to impersonate the data owner. Google Cloud also provides protection for phishing attempts on employees that have elevated access rights. In Tresorit, such impersonation attacks generally have very little impact due to the encrypted storage and lack of key storage/management in the cloud. However, less decryption operations can also mean faster data access depending on the AES mode of operation. GCM for example can be parallelized making decryption and encryption much faster.

In another storage service provided by Microsoft Azure named Azure Data Lake encryption is done with 3 keys: Master encryption key, Block encryption key and Data encryption key. The Master encryption keys are only rotated if they are managed by the user.[27] Same as above, the Data encryption key is used to encrypt the data and the master encryption key is used to encrypt the Data encryption key. The block encryption key is not stored anywhere, and it can only be calculated using the Data encryption key and the data block.[27] Azure Data Lake also provides data analysis on top of storage.

This makes identity management a very important component. In Microsoft Azure this can be done in multiple ways such as: multi-factor authentication, reverse proxy, single sign-on and azure RBAC [27]. Single sign-on gives the user the ability to sign in only once and mostly consists of storing certain cookies on the user's browser which makes it vulnerable to cross-site scripting and cross-site request forgery attacks depending on the browser. These types of attacks are also possible on Tresorit but the impact depends on what information is stored in the cookie, how is the certificate stored on the users machine and how is it related to the cookie, and whether or not key information is stored in said cookie. If all 3 types of information can be acquired, it would give the attacker the ability to log in and impersonate said user. If 2 or less types of information are gathered then the attacker would either not be able to impersonate the user, because he would not have the certificate required, or he would not be able to decrypt the data because he would not have the key. Two-factor authentication is generally safer, as the attacker needs to gain access to the authenticator on top of acquiring the password. Usually, the authenticator generates a one-time pad that is usable for a certain amount of time. In Microsoft Azure Single Sign on is done differently, using a private key owned by the user. The protocol works as follows [27]:

1. The client's device sends a Hello to the authentication server.
2. The server sends a nonce to the client that expires after 5 minutes.
3. The user signs the nonce with his private key and sends it back.
4. The signature is validated using the user's public key which is stored in a database located on the server. After this a token is generated that contains session key. The token is then encrypted with a transport key and sent back.
5. The user decrypts the response and can now communicate with the server using the session key. The key is protected by a module named Trusted Platform Module.

Depending on how the TPM module functions the user could also be vulnerable to targeted malware attacks that try to steal a user's private key. The protocol also heavily depends on what other Transport Layer protocol is used on top of it (ex: TLS), as in its current form it is vulnerable to man in the middle attacks because only the user proves his identity to the authenticator using an asymmetric key. Users can also authenticate themselves with a FIDO2 security key consisting of a USB device which is immune to phishing.[27] Phishing resistant authentication is also present in Google Cloud, although, only for very important employees.

It is uncertain whether the authentication is zero-knowledge or not. Because when the user logs in using either the single sign-on method or the FIDO2 key, both use a private key to sign a nonce, and the authenticator uses the respective public key to verify the signature. Most of the time there is a mathematical relation between a public key and a private key that allows an entity to gain certain information about the private key, although most of the time it is computationally hard to achieve this. All of this heavily depends on the algorithm and parameters used.

Azure RBAC is the system used to manage resources in Microsoft Azure. It defines 3 basic types of roles: an owner, a contributor, and a reader. The owner is the only person who can give access

permissions to other entities, the contributor can also modify resources, and the reader can only view them. It is also possible define custom roles.[27] If this basic model is applied in a scenario where multiple users are storing data and they want to share it and/or modify it between them it can lead to certain problems. The most inconvenient one would be the fact that every group member must gain an invitation from the owner. This problem is exactly what Tresorit tries to avoid in their modified I-TGDH structure so that new users do not have to depend on the availability of another user besides the inviter.

Microsoft Azure also takes extensive measures to avoid potential abuse situations on top of extensive logging practices. Staff may only gain access to a user's stored data temporarily and immediately revoked when the operation concluded. Third party contractors are also prohibited from using or reproducing anything [27]. It is not specified whether a user is notified when a certain staff member requires access to the data, and despite extensive logging access and strict access policies, the system remains vulnerable to abuse from external law organizations (such as police).

3.3.3 Encryption in Transit in Microsoft Azure

Just like encryption at rest, there are also multiple ways of securing the data in transit. As mentioned above one of those methods is TLS 1.2 and above. The Perfect forward secrecy problem is resolved using unique keys and RSA-2048 encryption keys to encrypt those unique keys.[27] The previously mentioned downgrade attacks can still be possible depending on implementation. Overall, this method of securing communications is inferior to Tresorit's Hardened TLS 1.3 and is similar to Google Cloud's modified TLS. Impersonating a member from a TLS session also has a similar effect as Google Cloud if the encryption is done on the server-side. If the encryption is client-side the effect is just like the one in Tresorit: the attacker cannot gain much as all the keys are not stored on the server. This problem can be easily solved by upgrading and using TLS 1.3 on all communication sessions. Enforcing this can be unappealing for certain customers as they would have to divert precious resources into upgrading the TLS. But such a change will have to take place eventually, as TLS 1.2 will probably become deprecated at some point in the future due to its many flaws and outdated algorithms. Another possible way of securing data in transit offered by Microsoft Azure is using the MACsec protocol (IEEE 802.1AE MAC Security Standards).[27] VPN protection is also present here. Some of these solutions can also be used together to increase the security (TLS + MACsec).

3.3.4 Conclusions

Overall, Microsoft Azure focuses on securing the cloud using firewalls, antimalware software, and extensive network layering alongside some cryptographic measure as opposed to focusing on more intricate cryptographic security solutions. This way, access to the data can be quicker and data processing can require less resources, while also remaining secure against outside attacks. This approach is likely chosen because of the large variety of services the cloud can provide. Just like Google Cloud, in the case of a security breach, it all depends on whether the attacker can compromise the key managing entity or not.

Microsoft Azure also services Microsoft OneDrive which is the out-of-the-box backup solution for Windows devices. This makes securing the cloud even more important as the number of devices running Windows is very large.

3.4 Amazon S3

Amazon S3 represents the cloud storage part of AWS (Amazon Web Services), and as the name implies, it is managed by Amazon. AWS also provides a large range of services (infrastructure as a service, platform as a service) just like the other two providers.

3.4.1 Infrastructure security and other aspects

Once again, physical security is present here as well. This layer is just as important as the others as physical hardware theft is always a potential problem. Even if an attacker is not able to break the encryption and decrypt the data, this scenario is one of the most potent DOS a certain individual can achieve. On top of this, the perpetrator can physically access all the hardware. Thus, just like the other services, Amazon performs constant surveillance over the cloud facilities, alongside logging all the traffic on the premises.[29]

In terms of storage decommissioning Amazon follows NIST 800-88 [29]. Microsoft Azure also follows this standard [27] while Google Cloud does not explicitly mention following the standard although the policy used has some similarities (data is physically destroyed if it cannot be erased). NIST 800-88 states different courses of action depending on the security rating of the information stored (this can be low, moderate, and high). If this rating is classified as low, then the hardware is never physically destroyed, but data recovery is considered unfeasible. In the moderate and high rating scenarios the hardware is always destroyed if the hardware will not be reused. Neither cloud vendor specifically mentions details about the rating process. Data is backed on multiple locations and multiple zones, so in case of multiple total failures it can still be recovered.[29]

Just like the other 2 services AWS uses certain modules [29] that separate the external network from the internal network in order to provide the same benefits mentioned in the other services. The modules are named API endpoints. The overall network architecture uses firewalls and ACL lists in order to easily filter out and deny access to unauthorized entities.[29] An outside user connects to the endpoint using SSL.[29] Network activity is constantly monitored in order to quickly abnormal behavior quickly and efficiently.

Two-factor authentication (called Multi-factor authentication) consists of a one-time 6-digit code that must be entered alongside the password and user. Users are forced to rotate their password every 90 days. Other credentials include Access keys, Key pairs, and X.509 certificates (only used for the Amazon S3 service).[29] It is not mentioned how the password is stored and whether the authentication is zero-knowledge is not. However, assuming the password is stored in a classic way (a secure hash function is applied over it), depending on the hash function, and the attacker's hardware, 90 days may not be enough to reverse the result. On top of this, if the user choses a very long password (100 characters) guessing it in

the 90-day time frame is also unfeasible. If there is no relationship between the password and other data used during the encryption/decryption no valuable information is gained from discovering an old password.

The hardware virtualization is done in a similar way to Microsoft Azure using a hypervisor [29] so that users cannot access each other's memory, storage etc.

3.4.2 Encryption at rest in Amazon S3

Amazon S3 gives the customer the ability to either use server-side encryption or client-side encryption. The same 3 options available in Microsoft Azure also exist here: server-side encryption with keys managed by the key management services, server-side encryption with keys managed by the customer, and server-side encryption with keys provided by the customer and managed by the key management services.[28]

In the first case scenario a single object is encrypted with an AES-256 key, and then the server encrypts the key with another key called a Master key. An object is the basic storage unit used in Amazon S3, being accompanied by some metadata that is not encrypted. Multiple objects are stored in another construct named Bucket. This model allows a much faster response than the other services when a user wants to access a specific file or object, because the data reconstruction step is either non-existent (because the data is not fragmented) or it is very fast (because it does not have many pieces). However, this way it is also much easier for an attacker to access the data for the exact same reasons, assuming he knows nothing about the keys forcing him to recover the master key to decrypt the object key to then decrypt the object itself. The data reconstruction step can be a large obstacle for an outside attacker (this is especially true if the data is split in many small pieces, like in Google Cloud). The option to create a bucket specific key also exists.[28] This key will be used to generate other keys for object encryption and is cycled at a certain time interval. During decryption, a customer master key (which exists in the key management service) is used to decrypt the bucket key and generate the object specific keys, making the process much faster while slightly reducing its security because all the object keys are now directly linked to one key.

In the third case scenario the services encrypt the object in the same way as above, except the key is removed from the memory as soon as the encryption finishes. Decryption also requires the same key from the customer. This verification step will be done by the key management service using a salted hash that was generated and stored when the encryption took place. This way the key managing entity cannot reconstruct the key and decrypt it [28] However, it is theoretically possible for the provider to view the data whenever the client wishes to retrieve it, because during this stage the data will be sent back in plaintext allowing the provider to see it. This model does bring some advantages over the Microsoft Azure version and Google Cloud because it does not allow the provider to decrypt the data whenever they wish to.

When using client-side encryption the customer also has 2 options: the option to use a key stored in the key management service located in the cloud, or to use and manage a locally stored key.[28] Because both the key and the data now reside in the cloud, the provider can, once again, theoretically decrypt the data whenever he pleases unless the encryption/decryption algorithm is not AES256. However, there are not many other encryption systems that provide the same level of security, efficiency, ease of

understanding, and speed as AES. Overall, this case scenario has the same downsides as the scenario where the key is managed by the customer and encryption is done by the provider.

Using the second option the customer must create a master key, and the client application will generate object specific keys. The object is then encrypted using this key, and the key is encrypted using the master key. All of this will be done by the Amazon client. The encrypted object is then sent to the provider for storage alongside some metadata for the key. During decryption, the metadata will be used to select the correct key [28]. If the master key is somehow lost by the customer, data cannot be recovered. Microsoft Azure does not provide many details about the client-side encryption. If the architecture for this scenario also requires the user to run a specific Microsoft Azure client, then both it and Amazon S3 are vulnerable to implementation level backdoors. If a client is not provided and the customer must build it himself, then the issue only exists in Amazon S3.

As long as the server key managing service manages the keys, they are guaranteed to be rotated at certain time intervals.

In 28 it is explicitly mentioned that the owner is the only entity that has access to the data he has stored in the cloud. The owner can also implement an access control policy in order to grant access to other users. These can either be user access policies, bucket access policies, or object access policies. Increased control over the data is generally beneficial for a customer, however, this comes with an increased risk of granting certain permissions to the wrong party because of potentials mistakes made when writing the policy. Activity logging at multiple levels is also present in this service, therefore it can be easy to tell if an unauthorized party has accessed the data because of a human error.

During backup data is also encrypted again using server-side encryption with customer master keys managed by the key management service. The servers also keep multiple versions of an object so that in case the current version is somehow lost from both the backup servers and the storage server, the user can still recover a previous version. The user also has the option to archive data or delete them if they are not used (this option is not present in the other services).

3.4.3 Encryption in Transit in Amazon S3

Access to the service is made using TLS 1.2 using Ephemeral Diffie-Hellman or Elliptic Curve Diffie-Hellman to guarantee perfect forward security. Every request must also be signed with a certain access key (AWS Signature V4 or AWS signatures V2) or use the AWS security token Service. TLS 1.0 is also supported (although not recommended) [28]. Just like Google Cloud and Microsoft Azure, a customer can also use a virtual private network to connect to the interface endpoint. AWS Signature V4 protects the data in transit by making sure it was not modified. The signature generated uses customer specific access keys which contain an access key ID and a secret access key. This can be considered an additional layer of protection on top of TLS.

3.4.4 Conclusions

In conclusion Amazon S3 prioritizes ease of use and access speed, as object encryption is faster than block encryption due to the amount taken to gather/split the files into pieces.

3.5 Overall conclusions

Table 2. Security Measure Comparison table

	Tresorit	Google Cloud	Microsoft Azure	Amazon S3
Encryption alg.	AES256	AES256/128	AES256	AES256
Data Sharing Method	I-TGDH	ACL	Azure RBAC	ACL
Type of Storage	Tresor[1]	Block	Block	Object/Bucket
# of keys required for decrypting one unit of data[2]	Highly Variable (at least 1)	4	2	2
Data in Transit Security Measures	Hardened TLS1.2	PFS-TLS1.2 /ALTS /VPN	PFS-TLS 1.2 /VPN /MACsec	PFS-TLS1.2 (TLS1.0 supported)
Encryption Location	Client-side	Server-side	Client-side /Server-side /Mixed	Client-side /Server-side /Mixed
Key-Managing entity	Client	Server	Client /Server /Mixed	Client /Server /Mixed
Key Rotation	Lazy re-encryption	90 days	Method not specified	90 days
Authentication	Zero-Knowledge	Standard/Not Mentioned	Standard/Not Mentioned	Standard/Not Mentioned
Other Authentication Security Features	PKI	Sign-in Location Tracking	X.509 certificates	Signature Based
Other Authentication Options	Two-Factor	Two-Factor /FIDO2	Two-Factor /FIDO2 /SSO	Multi-Factor

All the systems mentioned above fall into 2 categories: either the user has complete control over the data and the cloud cannot decrypt the information stored no matter what (unless explicit consent is given), or the provider can access the data at all times without notifying the user about the operation (the user consents to this by agreeing with the terms of service). Either case is also prone to abuse. In the first scenario the cloud can be used to facilitate illegal operations to which the cloud provider is oblivious to

[1] Tresorit uses Microsoft Azure servers to store data, so by association it also uses Block storage.
[2] Represents the number of unique keys required to decrypt one specific block/object/tresor.

because it cannot access the data. In the second scenario the provider can access the information whenever it wants and do whatever it wants. In both cases it is very hard for one party to prove that the other one has broken the terms of service in some way due to the lack of available information. However, it would be very hard to completely stop a provider from using the data in ways that break the terms of service while making sure it also retains the ability to moderate the files stored in the cloud. If the provider can view the files, it will also be able to create a local copy and work from there. The user also has the option to apply an additional layer of encryption over the stored files, denying the provider access.

Some possible solutions to the above problems include:

Threshold cryptography. The effect of this approach heavily depends on how the key shares are distributed. If the provider is able reconstruct the key by itself then the same effect as above is obtained. If the user is required to participate in the key reconstruction, he can deny access by simply refusing the participate (easiest way to achieve this is by disconnecting the device from the internet). The provider can threaten disciplinary action if the user refuses to participate. The harshest punishment consists of denying the user access to the data stored by deleting it. However, in a real case scenario if a user is aware of the sensitivity of the stored data, the punishment would end up helping the culprit erase evidence. Therefore, a great amount of effort must go into key shares management.

Encrypting the key with a weak encryption scheme and then storing it on the server, allowing it to mount an attack if it ever requires access to the files. The main difficulty here would be choosing a suitable encryption scheme so that the provider can complete the attack in a reasonably short amount of time. If the time required to finish the attack is too short, the provider can obtain the keys in large quantities ending up with the same problem discussed above. The hardware available to perform the attack also plays a fairly important role, making it even more difficult to find a suitable scheme.

Searchable encryption. At first glance searchable encryption seems to be the perfect solution. This approach would allow the provider to check the file contents for certain key words and determine whether they breach the terms of service or not. In practice this effect is much harder to obtain because users can either avoid use of sensitive words or they can use a certain coded language. Thoroughly investing a document could also take a long time as the appearance of one trigger word is not enough to properly judge whether the terms of service were broken or not. This is only made worse by files that are not documents (images, movies etc.). When searchable encryption is applied over such files, obtaining an accurate verdict is be very hard and it would take an even longer amount time, rendering this technique unusable when the provider has large amounts of files stored.

Explicitly requesting a user's permissions every time the provider wants to access the files. Similar to threshold cryptography, the user can disconnect himself and never respond to the request.

From the previous solutions we make the following observations:

- The file keys must be located on the server-side to guarantee file access to both the provider and the client at all times.
- The server must not have unlimited access to the files.
- The client must be notified when the server wants to view a file.

4 Proposed storage system

The proposed solution will use identity-based encryption (Cocks IBE) in order to force the provider to notify the user when a file is accessed and to provide the option of file-sharing (and to avoid using ACLs). This effect will be obtained using cryptographic methods. A log-based implementation can also be viable, although depending on approach, this variant could be prone to entities faking logs or other similar issues. The log in process will be zero-knowledge using the Fiat-Shamir Identification Scheme. This will deter the provider from attempting to reverse hash functions to gain the password, which in turn also grants it unlimited access to the data. For the average customer this is not a problem as the perks gained are likely not worth the effort. For high-profile customers with sensitive information this can be very important.

The keys will be stored in the cloud and the encryption and decryption operations will be performed locally. This approach discourages the client from breaking the terms of services because they will be inspected regularly. The user will also have the ability to tell whether the file has been viewed or not. On the other hand, the provider is able to access the files at all times, but he cannot do it covertly. Ideally, the terms of service would state a certain weekly/monthly/yearly inspection interval or a semi-fixed number of inspections over a time period, so that if the provider wishes to break the rule, the user can contest its decision and act accordingly (this can range anywhere from taking legal action to simply migrating to another provider). All of this means that the provider can still use the client information to create certain profiles with the purpose of manipulation, performance analysis and personalizing advertisements. As mentioned above this will always be the case in such a scenario (where the provider has access to the data). However, the approach chosen will inconvenience and even slow down such entities as they will not be able to constantly learn. If the provider wishes permanent access to a file, he will have to create a copy locally, which comes with additional concerns and costs.

Most of the cloud components are considered to be trusted but curious. The main downside of identity-based encryption is that the module that generates the keys also has access to them, meaning the entity managing this module would be able to decrypt all the files stored, on top of being able to perform other operations that can break the terms of service (such as changing the keys to deny the owners access to the files). Therefore, the key managing module and the file managing module must be trusted fully.

The architecture will follow a similar idea as the previously discussed cloud storage systems: there will be a network interface to facilitate traffic control. Because the network interface is trusted but curious, a separate encrypted connection will need to be established between the client and the key management module, so that all messages that are meant to go to the key management entity will also pass through the network interface. This is done to avoid a situation where an internal module creates a direct connection with an entity located outside of the local network. The contents of the messages that use this connection will either be an AES-256 key, an IV, or the Cocks private parameters, the size of whom also depends on the encryption algorithm chosen. The interface should be unable to decrypt such messages. Setting up the session keys can be done using any secure key exchange protocol that is resistant to man-in-the-middle attacks (TLS 1.3 is also suitable for both authentication and key exchange here). Because of this, the key

management module will also need to have its own dedicated virtual machine where the traffic will be scanned. Even though the size of the plaintext messages can be very small (32 bytes and 16 bytes) an attacker can attempt to insert a malware directly into the key managing entity because the network interface is unable to properly scan encrypted traffic.

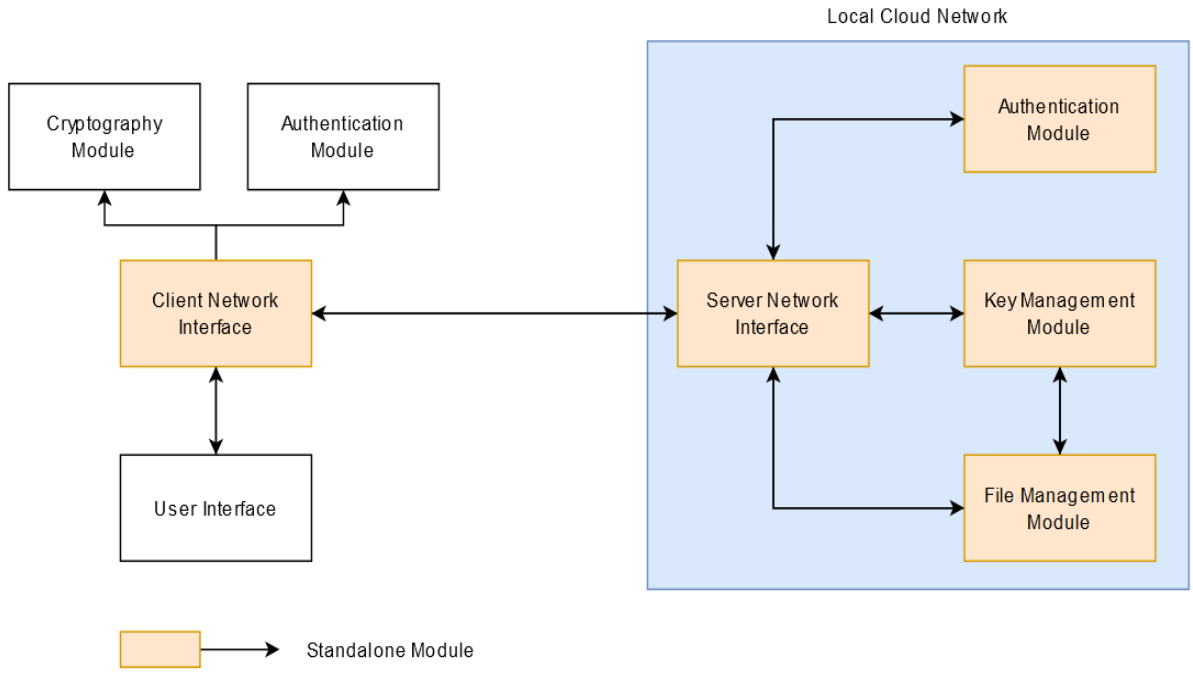

Fig. 3.3: Proposed storage system architecture

As it can be observed in fig. 3.3, the architecture has 5 active modules (named standalone modules in the diagram). These modules will communicate with other active modules using requests, similar to an API.

4.1 Operations

There are 13 operations that a customer can perform. However, the first thing the user should always do is create a secure TLS 1.3 session with the server. In this sub-section they will be explained in detail:

1. Register.
    1. The client generates p and q (1024 bits) using the authentication module and then calculates n. The user's password is used as a seed for generating s and v (both will contain 10 values), so that the login is possible. If the client did not use the password as a seed, the login step would be impossible because the values generated would be different every time. This also serves as a link between the password (which can be alphanumerical and have up to 32 characters) and the other values. This step can take up to ~3 seconds because p and q are large primes. If this generation took place on the server it could serve as a potential DOS attack.
    2. The client sends the registering request alongside n and v.

3. The SNI (Server network interface) receives the request and forwards it to CAM (cloud authentication module).
4. The CAM verifies if the user already exists in the database. If not, the user is inserted alongside n and v. It then responds to the SNI with the result of the operation.
5. If the operation was successful, the SNI sends a request to the KMM (key management module).
6. The KMM generates the Cocks private parameters and sends them to the user through the separate encrypted connection mentioned above.
7. The user encrypts the received parameters using AES-256 EAX (to ensure integrity) and then stores them locally. The key for this encryption will be the user's alphanumerical password.

2. Login. The client and the server will keep track of whether the user is logged in or not through a variable. On the client-side, some operations will be unavailable if the user is not logged in. The server also keeps track of this variable to make sure that the client does not perform illegal operations (E.g.: accessing someone else's files).
    1. The client sends a login request to the SNI containing his username.
    2. The SNI sends a request to the CAM and verifies if the username exists in the database.
    3. If the response is positive the SNI sends the a vector and the Fiat-Shamir identification process is performed a total of 4 times (this means that an attacker has a 2^40 chance to guess all the numbers correctly).
    4. If the result is correct all 4 times the client is successfully logged in. The login fails otherwise.
    5. Upon a successful login, the client attempts to load the secret Cocks parameters stored locally. If the EAX decryption fails, the user will be notified about a potential integrity violation. He must then attempt to reload the parameters to properly store and download files from the cloud.
3. Logout. The SNI maintains one thread for every other user (in order to avoid synchronization problems, the CAM, KMM and FMM maintain just one threat in total. When a user logs out the logged in value is changed. The thread is terminated when the user disconnects.
4. View Stored Files/View Shared Files. The KMM maintains a database of files and users that have access to them, and the FMM keeps track of owners and their files.
5. Reload Cocks Parameters. This option will be used if the parameters stored locally are corrupted/compromised. The client will first have to authorize himself successfully (even though he has already logged in) The decision to store them locally was made to minimize the number of messages between the client and the KMM, which in turn speeds up the response time.
6. Upload File. This option is only available to the file owner.

1. The client sends a message informing the SNI he wishes to store a file. He then generates a key and IV and encrypts said file with AES-256 CBC. Before sending the file for storage the client will save 2 hashes locally: one plaintext hash and one ciphertext hash. The file metadata will not be encrypted.
2. The client sends the file in segments of 4096 bytes alongside a hash to guarantee integrity.
3. The SNI waits until the file is successfully transferred, and then forwards it to the FMM.
4. Next the client will send the key and IV using the separated encrypted connection to the SNI which forwards them to the KMM.
5. The KMM encrypts the key using Cocks IBE and stores it. Additional keys will be created for every other user in the Access List. The KMM then informs the SNI about the status of the operation and then forwards the answer to the Client.

7. Download File. This option is only available to the file owner.
    1. The client sends a message to the SNI about the operation and the file chosen.
    2. The SNI verifies if the file exists in the KMM module. If it does, it is retrieved and sent to the client.
    3. The SNI then sends a request to the KMM module and forwards the encrypted response (containing the encrypted key and IV) to the client.
    4. The client decrypts his key and IV using the Cocks parameters stored locally. He then verifies if the ciphertext hash is the same as the one stored. If not, the application will inform the user that the file has been re-encrypted using a different iv (this means that the file has been accessed by someone else. After the files are decrypted, the hash is once again verified to check if the file was modified.
8. Delete File. File deletion is very simple. Once the request is sent to the SNI, it is forwarded to both KMM and FMM. The client is informed afterwards about the status of the operation.
9. View File Access List/ Update File Access List. The file access list is stored by the KMM. The client sends a username alongside the operations (remove/add user from access list). The KMM then either generates another user specific key or deletes one.
10. Download Shared File/ Upload Shared File.
    1. The client will send a request to the SNI.
    2. The SNI forwards the request to the KMM and FMM in this order.
    3. If the user who wishes to download file is not the owner, the KMM will first request the file from the KMM, re-encrypt it with the same key but different IV and then send it back to the KMM (in case of an upload operation, the KMM will re-encrypt the file after it is stored). After this, the KMM sends the key and the new IV to the SNI through the separate encrypted connecting, and the SNI forwards it to the user.
    4. In the case of a download, this is the step where the SNI receives the file. In the case of an upload, the user just receives the operation status as a response (succeeded/failed).

5 Final Conclusions and Future Work

Obtaining a compromise where neither side is favored is very difficult because the providers ability to view a file also implies, they are also able to copy the file. This can be potentially dangerous as any abuse is impossible to prove, and the prevention and punishment of such actions is at the providers discretion. There are many cryptographic cloud storage schemes that either offer complete privacy (or a high degree of it) for the user, or schemes that offer a high degree of control over the data for the provider (the storage system used by Google Cloud can fit in this category). The second type is much more commonly seen because providers prefer to moderate their data at their own discretion.

As for future work, replacement of IBE with something more decentralized would be an improvement to general user privacy, because the KMM component would no longer have access to all the keys. Another possible solution to this would be disallowing the KMM to communicate with the FMM module, this way even though the module knows all the keys he cannot decrypt any of the files. Anonymizing user identity/file names in a way such that the list of users/file names maintained inside the KMM is completely different from the one inside FMM, so that even though the KMM knows which key belongs to what file, he is unable to find out that specific file because it has a completely different id inside the FMM module. All these solutions come with one serious problem: the KMM is no longer able to perform re-encryptions which means that the encryption must either be performed in another module or on the client. This module would end up in the same situation the KMM was initially (it has access to all the keys, and all the encrypted file contents allowing him to view the data whenever he wishes) solving nothing. If the encryption is performed on the client-side, then the user can permanently deny a file inspection (easiest way to achieve this is to permanently disconnect the machine from the internet). In other words, solving this issue requires a completely different base mechanism.

Another possible issue appears when the file is shared with multiple users. Here, the owner will only be able to tell whether his file was accessed or not. He is unable to tell which user performed the operation. At first glance, this problem can be solved by tying an IV to a certain identity but, when using AES-256 CBC it is recommended to not use the same IV more than once (using the same IV more than once can lead to potential security issues). The user and the KMM module could share a pseudo-random number generator that generates IVs based on a certain string. The string will be a combination between the username and the current date. This approach requires the cloud to keep accurate logs. Sharing the pseudo-random generator should not be a problem as the IV does not need to be a secret.

If identity-based encryption is kept and a solution is not found (or finding a solution is deemed unnecessary) to the above-mentioned issues, Cocks IBE can be replaced for the full indent version of Boneh-Franklin IBE scheme. This would allow a much more efficient use of block storage. Cocks IBE can be used with block storage too, but a lot of space would be lost with key storage, as every AES key has 136kb after encryption, and block storage requires multiple keys (depending on block size and file size, the number of keys can even reach thousands).

6 Bibliography

[1] Sobti, Rajeev, and G. Geetha. "Cryptographic hash functions: a review." *International Journal of Computer Science Issues (IJCSI)* 9.2 (2012): 461.

[2] Eastlake, Donald, and Paul Jones. "US secure hash algorithm 1 (SHA1)." (2001).

[3] Krawczyk, Hugo, Mihir Bellare, and Ran Canetti. "HMAC: Keyed-hashing for message authentication." (1997).

[4] https://datatracker.ietf.org/doc/html/rfc5246 (Last accessed: December 2020)

[5] https://tools.ietf.org/html/rfc8446 (Last accessed: December 2020)

[6] https://www.ibm.com/support/knowledgecenter/en/SSB23S_1.1.0.14/gtps7/s7symm.html(Last accessed: December 2020)

[7] Bujari, Diedon, and Erke Aribas. "Comparative analysis of block cipher modes of operation." *International Advanced Researches & Engineering Congress*. 2017.

[8] Daemen, Joan, and Vincent Rijmen. "AES proposal: Rijndael." (1999).

[9] Goldwasser, Shafi, Silvio Micali, and Charles Rackoff. "The knowledge complexity of interactive proof systems." *SIAM Journal on computing* 18.1 (1989): 186-208.

[10] https://tresorit.com/blog/zero-knowledge-encryption/(Last accessed: December 2020)

[11] Fu, Kevin Edward. *Group sharing and random access in cryptographic storage file systems*. Diss. Massachusetts Institute of Technology, 1999.

[12] Kallahalla, Mahesh, et al. "Plutus: Scalable Secure File Sharing on Untrusted Storage." *Fast*. Vol. 3. 2003.

[13] Bellare, Mihir, Ran Canetti, and Hugo Krawczyk. "Keying hash functions for message authentication." *Annual international cryptology conference*. Springer, Berlin, Heidelberg, 1996.

[14] Kravitz, David W. "Digital signature algorithm." U.S. Patent No. 5,231,668. 27 Jul. 1993.

[15] Kerry, Cameron F., and Charles Romine Director. "FIPS PUB 186-4 federal information processing standards publication digital signature standard (DSS)." (2013).

[16] Szebeni, Szilveszter, Levente Buttyán, and István Lám. "Method and system for handling of group sharing in a distributed data storage, particularly in P2P environment." U.S. Patent No. 9,563,783. 7 Feb. 2017.

[17] https://tresorit.com/security/encryption (Last accessed: February 2021)

[18] Tresorit whitepaper

[19] Szebeni, Szilveszter, and Levente Butty'n. "Invitation-oriented TGDH: Key management for dynamic groups in an asynchronous communication model." *2012 41st International Conference on Parallel Processing Workshops*. IEEE, 2012.

[20] Kim, Yongdae, Adrian Perrig, and Gene Tsudik. "Tree-based group key agreement." *ACM Transactions on Information and System Security (TISSEC)* 7.1 (2004): 60-96.

[21] Szebeni, Szilveszter, and Levente Butty'n. "Tresorium: Cryptographic file system for dynamic groups over untrusted cloud storage." *2012 41st International Conference on Parallel Processing Workshops*. IEEE, 2012.

[22] Cuppens, Frédéric, Nora Cuppens-Boulahia, and Julien Thomas. "S-TGDH, secure enhanced group management protocol in ad hoc networks." *CRiSIS'2007: International Conference on Risks and Security of Internet and Systems, colocated with IEEE GIIS*. 2007.

[23] Rosenberg, Jonathan. *Interactive connectivity establishment (ICE): A protocol for network address translator (NAT) traversal for offer/answer protocols*. RFC 5245, April, 2010.

[24] Liu, Fang, et al. "NIST cloud computing reference architecture." *NIST special publication* 500.2011 (2011): 1-28.

[25]Mell, Peter, and Tim Grance. "The NIST definition of cloud computing." (2011).

[26] Google Cloud Security Whitepapers, March 2018 (https://services.google.com/fh/files/misc/security_whitepapers_march2018.pdf?hl=ar)

[27] Microsoft Azure Security Documentation https://docs.microsoft.com/en-us/azure/security/fundamentals/ (last visited February 2021)

[28] Amazon S3 documentation https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html (last accessed March 2021)

[29] https://d1.awsstatic.com/whitepapers/aws-security-whitepaper.pdf (last accessed March 2021)

[30] Cocks, Clifford. "An identity based encryption scheme based on quadratic residues." *IMA international conference on cryptography and coding*. Springer, Berlin, Heidelberg, 2001.

[31] Micali, Silvio, and Adi Shamir. "An improvement of the Fiat-Shamir identification and signature scheme." *Conference on the Theory and Application of Cryptography*. Springer, New York, NY, 1988.

[32] R. Kaur and A. Kaur, "Digital Signature," *2012 International Conference on Computing Sciences*, Phagwara, 2012, pp. 295-301, doi: 10.1109/ICCS.2012.25.