

End-to-End Encryption

- Abstract
- 1. Introduction
- The application
- 2. Theoretical Aspects
- Symmetric key encryption
- Public key encryption
- Attack types
- Authentication
 - Message authentication codes
 - PRFs
 - HMAC
 - Authenticated encryption
 - Authenticated encryption with associated data
- Digital signatures
- End to end encryption
 - Drawbacks
 - Metadata
 - Man-in-the-middle attacks
 - Endpoint security
 - Backdoors
- AES
- Classical Diffie Hellamn
- Elliptic curve cryptography
- 3. Existing Technologies
- Signal protocol
 - Diffie Hellamn
 - Extended Triple Diffie Hellamn

- Double Ratchet
 - EdDSA signature
 - Sealed sender
 - Security analyses
 - Whatsapp
- MTPProto
 - General description
 - Security analyses
- Signcryption
 - Signcryption
 - Security analyses
- Letter Sealing
- Threema
- Group messaging
- About MLS?
 - MLS
- 4. Technologies used
- 5. The application
- General flow
- Implementation and code layout
- 6. Conclusions
- 7. References

Abstract

End-to-end encryption is intended to prevent data from being read by other parties, i.e. servers, service providers, the government, than the ones directly involved into a conversation. This approach has gained popularity in recent years in instant messaging applications and many of these have developed their own protocol or adapted a base one.

The aim of this thesis is to present a synthesis of the state-of-the-art of the most popular technologies, as well as the previous vulnerabilities present either in the protocols themselves or the applications implementing them.

Section 2 introduces theoretical concepts and algorithms that are fundamental to the protocols. In Section 3, each of the end-to-end encryption protocols, namely Signal, MTProto, Signcrypt, Letter Sealing and Threema, are discussed in more depth, along with known vulnerabilities and limitations. Moreover, the problem of end-to-end encrypted groups is analysed along with the efforts in creating an efficient and secure layer for group messaging.

Section 4 and Section 5 contain information about the application and the technologies used in its development. The intent of the application is to show, in a comprehensive way, the benefits and the drawbacks of using end-to-end encrypted messaging applications. Along the regular features of an instant messaging application, it will provide a “third party view” feature that shows what information is available to a third party.

Section 6 Is reserved for the conclusions.

1. Introduction

[1]: iMessage

[2]: Whatsapp whitepaper

[3]: Messenger Starts Testing End-to-End Encryption with Secret Conversations

[4]: Signal

[5]: Trevor Perrin - The XEdDSA and VEdDSA Signature Schemes, 20.10.2016

[6]: Trevor Perrin, Moxie Marlinspike - The X3DH Key Agreement Protocol*, 04.11.2016

[7]: Trevor Perrin, Moxie Marlinspike - Double Ratchet Algorithm, 20.11.2016

[8]: Moxie Marlinspike, Trevor Perrin - The Sesame Algorithm: Session Management for Asynchronous Message Encryption, 14.04.2017

As online communication gained more popularity, using instant messaging

applications become a standard in our quotidian lives. Therefore, the need for assurance that there is no third party spying on our conversations, either the government, the service provider or an attacker with malicious intent, grew even more, especially in states where free speech is threatened.

End-to-end encryption is used to protect the privacy of the messages sent between two or more participants, as they are in transit or at rest, with the intended recipients being the only ones that can decrypt and read the messages. Thus, the third parties interested in intercepting the data sent are unable to see the actual plaintext and there is no need for the users to put their trust in them. Moreover, the messages' contents are protected against possible data breaches.

In case the service providers, compliance with regulations regarding data privacy and security is easier [?].

But the cost of benefits is paid with a number of limitations. Some of these are user metadata collection, which can be used for advertising[?]. This issue was tackled by Signal with its Sealed Sender feature[?].

Also, spam and abuse becomes harder to control on end-to-end encrypted platforms and backdoors are heavily requested by governments in order to protect the citizens and investigate criminal activities[?]. More will be discussed in Section 2, along with a brief presentation of the cryptographic concepts used throughout the thesis.

Messaging apps using end-to-end encryption have been around since 2012, with iPhone's native messaging app iMessage[1] and then Signal[4], previously known as TextSecure, developed in 2013. This practice became more popular in 2016, when Whatsapp announced the introduction of end-to-end encryption by default in the application[2] and Facebook Messenger added the "secret chats" feature with similar properties[3].

The issue with group chats is also addressed. Currently, each application have either rolled out their own implementation[?] or end-to-end encrypted group chats are not supported at all. [?] The current implementations are not very practical and tend to be more complex as the number of supported members grows, so an idea of using a security layer for this purpose emerged

in 2016 [?], called Messaging Layer Security.

The purpose of this thesis is to create an overview of the most used end-to-end encryption protocols in popular instant messaging apps and their mechanisms. The topics that will be discussed are their current implementations and innovations in the field, as well as past security issues. The protocols analyzed are Signal, MTProto, Signcrypton, Letter Sealing and Threema, in Section 3. Section 6 will sum up the ideas and findings presented in this thesis and during the implementation of the application.

The application

The application is a web-based instant messaging application which provides end-to-end encryption, by default, for both private and group chats. The users register with an email and can send messages and attachments.

In order to illustrate the risks of not using end-to-end encrypted applications, the user can initiate unencrypted chats with other users. To have a better view of the implications of this, the user can choose to see through a “third party’s view”. This will show what a third party can see, including metadata and the messages (encrypted or unencrypted, depending on the user’s choice).

The implementation and the used frameworks and libraries are discussed at large in Section 4 and Section 5.

2. Theoretical Aspects

- might be useful for this section

In this chapter, some of the main concepts that will be used throughout the thesis are briefly presented.

Symmetric key encryption

- HOAC 33

- https://en.wikipedia.org/wiki/Symmetric-key_algorithm
- <https://resources.infosecinstitute.com/topic/padding-oracle-attack-2/>

Symmetric-key encryption is an encryption scheme which uses the same key for both encryption and decryption. In this case, the key must be a shared secret between the communicating parties, which might result in security issues if the key is intercepted, if it is sent through an insecure channel.

An advantage of this type of algorithms is that they are more efficient in terms of software and hardware.

- /e image

One such cryptographic algorithm that is popular in end-to-end encryption protocols is AES (Advanced Encryption Standard).

Public key encryption

- HOAC 43, 301
- https://en.wikipedia.org/wiki/Public-key_cryptography
- <https://resources.infosecinstitute.com/topic/padding-oracle-attack-2/>
- enc function - trapdoor function with the nec info being the decryption key

Public-key encryption, or asymmetric encryption, is an encryption scheme which uses a public and a private key pair for each user. The public key is known and can be publicly distributed, so sending it through an insecure channel is not an issue anymore, but the private key must be kept secret by the user. To encrypt a message, the sender uses the public key of the receiver, which can be decrypted only using the recipient's private key.

The security of this encryption scheme resides on the property of the key pair that, while knowing the encryption key, it must be computationally infeasible to obtain the plaintext message from a random ciphertext, so obtaining the decryption key.

A drawback of public key encryption is that it is less efficient than symmetric key encryption but it can be used as a secure channel for key exchange or for

encrypting smaller data sets.

Examples of asymmetric key algorithms, commonly used in popular end-to-end protocols, are the Diffie-Hellman key exchange protocol and Elliptic curve based cryptography.

Attack types

- https://www.tutorialspoint.com/cryptography/attacks_on_cryptosystems.htm
- <https://www.ics.uci.edu/~stasio/ics8-w12/Week9%20-%20part%202%20-%20Symmetric%20Key%20Encryption.pdf>
- <https://www.cs.clemson.edu/course/cpsc424/material/Cryptography/Attacks%20on%20Sym>

There are various attack vectors that can be identified on cryptosystems. Their aim is to obtain the private key based on information that can be collected from plaintext or ciphertext, usually. Some of them are briefly presented below. Also, vulnerabilities to some of these have been present in the protocols that will be analyzed further.

The **ciphertext only attack** involves an attacker who has access to the ciphertexts and can be successful when the plaintext can be obtained from a set of ciphertexts. The encryption key can be determined afterwards.

Modern cryptosystems are not affected by this type of attack anymore.

In the **known plaintext attack**, the attacker knows the plaintext and corresponding ciphertext and its aim is to obtain the key. This type of attack works on simple ciphers.

Side channel attacks are more concerned on the way the computer system is implemented, not on the implementation of the algorithm itself. Usually physical, the information that can be used against a cryptosystem consists of power usage, the amount of time the process takes, sounds or electromagnetic radiation leaks.

Impersonation

An adversary can place themselves in the communication between two parties, A and B, and send their public key such that A thinks it was B's public key.

In this way, the adversary can decrypt the message, read and/ or alter it before encrypting it with B's key and sending it forward.

This kind of attack can be mitigated using authentication, so guaranteeing that the recipient is the intended one.

Chosen plaintext attack - wiki - wiki, semantic security

The adversary chooses arbitrary plaintext and then is given the corresponding ciphertext. The intention is to reduce the security of the encryption scheme.

This attack can be classified further into batch attacks and adaptive attacks. In case of a batch chosen plaintext attack, the adversary knows the plaintext before seeing the corresponding ciphertext, while in the adaptive chosen ciphertext attack, the attacker can request more ciphertexts after seeing the ciphertext of corresponding plaintext. /x

This vulnerability can be fixed by providing semantic security, meaning that the adversary should not be able to derive anything but negligible information about a plaintext message, given the ciphertext and the public key. This property is also called indistinguishability under chosen plaintext attack.

Chosen ciphertext attack - wiki

In this type of attacks, the adversary has access to the decryptions of chosen ciphertexts and the intention is to obtain the private key.

It can be split in two categories, too: indifferent or "lunchtime" attack and adaptive attack. The lunchtime attack refers to the fact the the attacker can receive decryptions of any chosen ciphertext until a certain point. The term comes from the fact that the attacker has access to the user's device that can decrypt these ciphertexts, which was left unattended. In adaptive chosen ciphertext attacks, the adversary is only allowed to choose ciphertexts related to the target one and obtain enough information to decrypt it.

To avoid such attacks, the cryptosystem should not provide any decryption oracles, for example.

Brute force attacks are the slowest ones. They are done by trying all the possible keys until the message is decrypted, hence the decryption key is

obtained.

Authentication

- intro - hoac 42 [1]
- identification and entity auth - hoac 401 [2]
- contemporary crypto/ 318 [3]
- <https://en.wikipedia.org/wiki/Authentication>

Authentication is the process of proving the identity of an entity, called claimant, to a verifier and preventing impersonations. It might be done using certain credentials (a password) or with a digital certificate (in case of websites).

The process of entity authentication finishes with acceptance or rejection. Therefore, for three distinct parties, A, B honest and and a third party C, the following objectives can be defined: [2] - if A successfully authenticates to B, then B accepts the identity of A - after a successful identity exchange between B and A, B cannot impersonate A to C - if C tries to impersonate A, there is a negligible change that B will authenticate A, even after a large number of authentication protocols between C and A and B.

Another form of authentication is data origin authentication or message authentication. These are techniques that assure one party of the identity of the sender. Usually, the message has additional information attached so the receiver can determine it.

According to [3], there are two types of technologies used for message authentication: - digital signatures, based on public key cryptography, where the private key is utilized when generating the digital signature and the public one is the used for verification - message authentication codes (MACs), using symmetric keys

Message authentication codes

- Serious crypto pg 179 [1]

- contemporary crypto/ 318 [2]
- ? MLS
- keyed hashing functions (hashing func with secret keys) => message auth codes (MAC) and pseudorandom func (PRF)

MACs (Message authentication codes) are authentication tags created from the message and the secret key and protect the authenticity and integrity of the message. The secret key must be known by both participants because it is also used to verify the MAC and to confirm that a message was not modified in transit.

They are usually combined with a cipher and, in this way, the message's integrity, authenticity and confidentiality is kept.

Attacks

A MAC is secure when the attacker cannot create a tag for a message when they don't know the key. In [2], two types of attack vectors are presented: forgery attacks and replay attacks.

- forgery - create a tag when you don't know the key
- attack vectors:
- known-message attack - tags and data collected by an eavesdropper
- chosen message attack - the attacker chooses the messages to be auth and if the attacker is able to adaptively choose other messages and their corresponding MACs, it is an adaptive chosen-message attack
- replay attacks - capture a message and resent it to the receiver, pretending to be the sender - mitigation by numbering the messages?
- It can be concluded that if the adversary can determine the secret key, the system is totally broken. If the MAC of a meaningful message was determined, then the MAC is selectively forged, and if the message is not meaningful, the MAC is existentially forged. ? rephrase
- A way to lower the possibility of guessing a MAC (which is always possible?) is to increase the tag length and a message authentication

system is secure if the attacker can only guess the MAC in order to forge it.

- Another issue is whether the attacker can verify the guessed MAC and this gives us two new categories: verifiable and non-verifiable MACs.
- Non-verifiable MACs are more secure, since it's not possible to find the correct one using brute-force.

PRFs

- pseudorandom functions sc/ 181 [1]
- contemporary crypto/ 327 [2]

Pseudorandom functions turn a message, using a secret key, into a seemingly random output. As defined in [2], a function $f : X \rightarrow Y$ is pseudorandom if it is randomly chosen from the set of all mappings from X to Y .

They are considered stronger because the requirement that the output is indistinguishable from a random string is stronger than the inability to forge tags. [1]

- not meant to be used on their own
- key derivation schemes use this to generate crypto keys from a master key or password
- ident keys use this to generate a response from a random challenge? - ex a server sends a random challenge message and the recv should prove with this that it knows the key
- tls - prf to generate key material from a master secret and session specific random values

HMAC

- sc pg 184 [1] attacks and info about cbc-macs/ 186
- <https://datatracker.ietf.org/doc/html/rfc2104> [2] + formula

Hash-based MAC is a MAC which is obtained from a hash function, function

that produces a fixed-size hash value out of random-sized data and they should be collision free, and are used by the end-to-end encryption protocols that will be analyzed later. Therefore, to compute it, one needs a cryptographic hash function, a secret key and the message. Similar to MACs, they are used to prove authenticity and integrity of the messages and their strength depends on the strength of the hash function.

Authenticated encryption

- sc 200
- wiki
- contemporary crypto/ 451 - entity encryption

Authenticated encryption (AE) is a mix between a cipher and a MAC and it is used to assure data confidentiality and authentication.

The combinations are different in terms of the order of the encryption and authentication, and the following three ways can be defined. The message is not accepted if the ciphertext or the tag was corrupted.

Encrypt and MAC The ciphertext and the tag are computed separately. The recipient then decrypts the ciphertext and uses it to obtain the tag and compares it to the received tag.

MAC then encrypt The tag is computed first and it is concatenated to the message, which is encrypted afterwards. The recipient decrypts the ciphertext and computes the tag from the resulting plaintext.

Encrypt then MAC The tag is obtained from the ciphertext and the recipient computes the tag and only if the received and obtained tags are equal, the message is decrypted.

Authenticated encryption with associated data

- sc 204, AES GCM/ 206

A version of AE is the authenticated encryption with associated data (AEAD).

The authenticated data is processed by an authentication cipher but it is kept in plaintext. This is useful if you need certain data to be available, such as a header, but the payload needs to be encrypted.

Authentication ciphers use a secret key and a message to obtain the ciphertext and the tag together, making the process faster and more secure than the normal AE. The decryption phase uses the ciphertext and tag along with the key to obtain the plaintext and authenticate the data.

The output of an AEAD operation is the ciphertext, tag and the unencrypted associated data, obtained from the key, plaintext and same associated data. Thus, the tag depends on both the plaintext and associated data. In order to decrypt and verify the message, the key and output parameters are needed.

Moreover, if the plaintext is empty, the algorithm can be considered a normal MAC. Similarly, if the associated data is missing, it becomes a normal authentication cipher.

Digital signatures

- intro - hoac 40
- digital signatures (ch 11) - hoac 441
- contemporary crypto/ 396
- https://en.wikipedia.org/wiki/Digital_signature
- <https://blog.pandadoc.com/what-is-a-digital-signature-and-how-does-it-work/>
- <https://www.docusign.com/how-it-works/electronic-signature/digital-signature/digital-signature-faq>
- <https://cybersecurity.att.com/blogs/security-essentials/digital-signatures-101-a-powerful-and-underused-cybersecurity-ally> /////

Digital signatures are values that bind the identity of the originating entity to the contents of the message or document. They are used to verify the authenticity and integrity of the messages and to provide non-repudiation, meaning that the signer cannot successfully claim that they did not sign the message.

The digital signature scheme is similar to public key encryption, and consists of the following algorithms: **Key generation** A public and a private key are generated. The private one is kept secret, while the other one is publicly available. **Signing process** The signature is produced using the private key of the signer and the message. **Signature verification process** From the public key of the sender, the message and the signature, the authenticity of the message can be either accepted or rejected.

They must be correct (the valid signatures must be accepted) and secure (it should be computationally infeasible to forge a signature without knowing the key). Also, it must be computationally infeasible to find a valid signature for a message and a verification key without knowing the signing key.

Attacks - cc/397 [1]

There are two major attack vectors that are taken into account regarding the security of a digital signature: - key-only attack, when the attacker knows the signatory's verification key but has no information about the signed messages - message attack, when the attacker knows both the signatory's verification key and some information about the messages or is able to obtain it

The second category can be further split into the following attacks: - known message attack, when a certain number of unchosen messages and their corresponding signatures are known - generic chosen message attack, the signatory's key knowledge is independent of the attacker's choice of messages (along with their corresponding digital signatures); the messages must be chosen a priori - directed chosen message - similar to the previous attack vector but this one is directed against a signatory's key - adaptive chosen message - the attacker can obtain the digital signatures of a chosen list of messages and it depends on the signing key; the list of messages can be adaptively chosen during the attack (the attacker has access to the signature generation oracle => for every message, the attacker returns a valid signature)

- maybe about the security breaks?
- total break - determine the key
- universal forgery - find similar signature algo
- selective forgery - forge digital signature for a particular message, chosen before

- existential forgery - forge digital signature for at least one random and not necessarily meaningful message

End to end encryption

- Wiki
- Protonmail e2ee ??
- Encryption in-transit and Encryption at-rest – Definitions and Best Practices
- Data Protection: Data In transit vs. Data At Rest
- Brief presentation about ee2e
- What end-to-end encryption is, and why you need it
- What is End-to-End Encryption?
- end-to-end encryption (E2EE)
- A Deep Dive on End-to-End Encryption: How Do Public Key Encryption Systems Work?
- WhatsApp, Signal and End-To-End Encryption
- using quantum key distribution
- efficient
- <https://www.theitstuff.com/what-is-end-to-end-encryption> - algo
- <https://infosec-handbook.eu/blog/limits-e2ee/> to read

End-to-end encryption is a communication system in which the messages can be read only by those participating in the conversation and is allowing them to securely communicate through an unsecured channel.

The general algorithm is based on public key cryptography. The data is encrypted by the sender, at the endpoint, using the public key of the receiver and the only way to decrypt it is by using the recipient's private key. This

ensures that the data cannot be read or modified by any third party involved, since they don't have access to the private keys.

The need for this method arises from the fact that many messaging applications use third parties to store the data and it is protected only "in transit" (ex: TLS), meaning that it can be decrypted and read and/ or tampered with before redirecting it to the recipient, when it reaches the server. Therefore, the privacy of data and the user is put at risk, since the contents can be used and interpreted by anyone with access to the server.

Drawbacks

Metadata

[1] <https://www.whatsapp.com/legal/updates/privacy-policy/?lang=en> [2] <https://www.bbc.com/news/technology-55634139> [3] <https://www.androidauthority.com/whatsapp-privacy-change-delay-1223909/> - to read

An important drawback of end-to-end encryption is that metadata about the users or messages can be collected and it is accessible to the server. This information includes the time at which the user is online and for how long, when the message was sent, to whom and so on and information about the device. This data can be used to track the users' activity or be sold to advertising companies.

An example of the impact of metadata collection is Whatsapp's update of terms and services[1] in 2020. They announced that information regarding the location, browser information, device hardware and connection etc. is automatically collected. This resulted in a shift of the users to other applications considered more secure, like Signal and Telegram. [2]

Some apps have implemented ways to collect as little metadata as possible. These will be addressed later, in Section 3.

Man-in-the-middle attacks

In this type of attacks, the attacker is needs to inject themselves between two

endpoints and to impersonate one or more of the participants. The sender will unknowingly use the public key of the attacker who can now read or alter the messages before they are forwarded to the original recipient.

This can be avoided if the participants' identities are verified. Some applications provide authentication via QR code scanning or using safety numbers.

Endpoint security

The messages are only protected from possible eavesdroppers on the communication channel or while the data is at rest, but the endpoints are still vulnerable. After decryption, the messages in plaintext are available to anyone who has access to the endpoint device, so they can be accessed using other methods (ex. device theft, social engineering, hacking the device).

Backdoors

[1] <https://www.justice.gov/opa/pr/international-statement-end-end-encryption-and-public-safety> [2] https://www.politico.eu/wp-content/uploads/2020/09/SKM_C451_new.pdf [3] <https://data.consilium.europa.eu/doc/document/ST-12863-2020-INIT/en/pdf> [4] https://www.boxcryptor.com/en/blog/post/e2ee-weakening-eu/?utm_medium=post&utm_source=newsletter&utm_campaign=en.newsletter.b2bb

The service providers might include, intentionally or not, ways to access the data by bypassing the encryption. These are called backdoors and have been highly requested by governments across the years. These weaknesses are mostly needed in order to protect public safety and to “protect citizens by investigating and prosecuting crime and safeguarding the vulnerable” [1].

An example might be iMessage's iCloud backup function, but needs research <https://news.ycombinator.com/item?id=25078034> <https://news.ycombinator.com/item?id=250782> <https://www.iphoneincanada.ca/news/imessage-security-flaw/>

<https://www.cbsnews.com/news/paris-attacks-encrypted-messages-does-the-government-need-a-way-in/>

AES

- contemporary crypto/ 282
- NIST paper
- diagrams for the algo <https://proprivacy.com/guides/aes-encryption>
- <https://techjury.net/blog/what-is-aes/>

AES (Advanced Encryption Standard) is a symmetric block cipher, based on a substitution-permutation network, which uses keys of length 128, 192 or 256 bits to process data in blocks of 128 bits, introduced by NIST in 2001. It was approved by the NSA and is widely used in modern applications, due to its efficiency and security.

About the algorithm

The operations are performed on a State array, which is a two-dimensional array of bytes, having the block length div by 32, Nb (Nb = 128 / 32 = 4) rows and columns and the cells contain 1 byte of the block. In the beginning, the input bytes are copied into the State and after the encryption/ decryption, the output bytes are copied again in the State, row by row (figure at pg 285)

- the key length is computed in the same way, so it would have length Nk 4, 6, 8, depending on number of columns in the cipher key (from 128, 192, 256 bits)
- the state array can be interpreted as a state array of 32 bit words
- operations over a finite field
- addition - xor
- multiplication - multiplication of the polynomials mod irreducible polynomial $x^8 + x^4 + x^3 + x + 1$
- round keys are values derived from the cipher key and are applied to the state
- these values are fixed: Nr = 10, Nk = 4; Nr = 12, Nk = 6; Nr = 14, Nk = 8
- the round function is composed of:

- byte subst using a subst table (S-box)
- shift the rows of the state by different offsets
- mix data in each column
- add round key to state
- after an initial round key addition, the round function is implemented $N_r - 1$ times
- the inverse cipher (decryption) is following the previous steps but in reverse order
- algo + sbox/ 287 and so on with the rest of the operations

Modus operandi - won't write too much about these, maybe just the CTR mode - 296 - electronic code book mode (ecb) - plaintext m is split into t n -bit blocks and a padding is added if there are not enough bits and encrypted - prb: same ciphertext blocks everytime, same key, same message; no protection - cipherblock chaining mode (cbc) - encr depends on the key, all prev message blocks and an initialization vector that is public \Rightarrow identical plaintext blocks are mapped to different ciphertext blocks - prb: w/ the init vector, the ciphertext is 1 block longer; if an error occurs in one block, the error is propagated (ex transmission errors) - cipher feedback mode (cfb) - turns block cipher into stream cipher by generating a seq of pseudorandom bits from the block cipher that are added mod 2 to the plaintext to produce the ciphertext bits - good when you don't need large data transfers - prb: performance; error propagation; can't precompute key stream because you need the ciphertext block - output feedback mode (ofb) - similar to cfb - no error propagation - it is important to change the init vector regularly if you have the same key - counter mode (ctr) - taken from serious crypto/ 111 - turns a block cipher into a stream cipher - enc blocks composed of a counter and a nonce - counter - integer that is incremented for each block - unique for each block - nonce - number used only once, each message has a unique nonce - enc: xor the plaintext and stream (obtained by "enc"? the nonce and the counter)

Attacks - https://www.schneier.com/blog/archives/2009/07/new_attack_on_a.html

- https://www.schneier.com/blog/archives/2009/07/another_new_aes.html
- <https://blogs.ntu.edu.sg/science/2021/03/03/finding-keys-to-the-most-secured-encryption-algorithm/> - side channel attacks are possible

aes: NIST, Announcing the ADVANCED ENCRYPTION STANDARD (AES)

Classical Diffie Hellamn

- serious crypto/ 268 [1]
- <https://www.cs.jhu.edu/~rubin/courses/sp03/papers/diffie.hellman.pdf> [2]

Diffie Hellamn (1976)[2] is a key agreeent protocol that allows the participants to share a secret between them, with the exchanged information being public. The secret is turned into session keys and used as symmetric keys to encrypt and authenticate data or to be used as a secure channel, during the session[1 pg 273].

The algorithm

The mathematical function involves a big prime number p and a base number/ generator g as public information, and a number from the Z_p^* set, chosen by each participant, which is kept private.

To illustrate the algorithm, for two participants we have the numbers a and b . Then each of the participants computes $A = g^a(mod)p$, $B = g^b(mod)p$ and makes these computations publicly available. The other participant takes this result and raises it to their private number and this will be the shared secret, so: $(g^a mod p)^b = (g^b(mod)p)^a = g^{ab}(mod)p$.

Security and attacks

The security of the Diffie Hellman protocol resides on the discrete logarithmic problem, which means that you need to recover a from $g^a(mod)p$; this is possible for smaller values, but it is infeasible if the values are chosen correctly.

- attack models, as named and defined in [1 / 275]
- the eavesdropper - attacker observers the message exchange and can

record, modify, drop, inject messages; to protect against, the protocol should not leak info

- the data leak - the attacker has access to some of the session keys and temp secrets (from one or more exec of the protocol), but not the long term secrets
- the breach - the attacker knows the long term key of one or more parties
- security goals
- authentication - auth key agreement - both parties auth
- key control -
- *forward secrecy* - if the session is compromised, the keys used in the past cannot be compromised
- resistance to key-compromise impersonation - happens when the long-term key is compromised
- *backward secrecy* - if a session is compromised, the following communication is not compromised if at least one uncompromised message is sent between the parties

extra - non-dh key exchange: 3g and 4g communications with sim cards

Elliptic curve cryptography

- serious crypto/ pg 288 [1]
- the presentation saved somewhere [2]
- <https://scholar.rose-hulman.edu/cgi/viewcontent.cgi?article=1101&context=rhumj> [3]
- /e needed
- they are curves which are also groups, so they keep the group axioms

- they can also be defined over finite fields and the law is constructed geometrically, the points on the curve with equation $y^2 = x^3 + ax + b$ (Weierstrass curve) - but can have different forms
- they are horizontally symmetric
- the operations are addition and multiplication ++
- general idea of addition (geometrically): ++ pictures
 - fix the two points on the curve and draw a line through them until it intersects the curve again
 - the sum of the two points is the reflection of the second/ third point
 - if the points are the same, draw the tangent through that point until it intersects the curve again and the result is still the reflected point
- multiplication: add a point to itself multiple times - this can be optimized using the double and add method
- combined with DH => DH key agreement over elliptic curves
- digital signatures (ECDSA)
- encryption
- NIST curves: standardized curves and one that is throughoughly used is Curve25519 and has the equation $y^2 = x^3 + 486662x^2 + x$, which works with numbers mod $2^{255} - 19$ (256 bit prime number)

The algorithm

To exchange a shared secret over an insecure channel using this protocol, the algorithm is similar to the one of the classical Diffie Hellman protocol.

The public variables are the base point P and the elliptic curve over a finite field $E(F_q)$. The participants need to choose a random integer k_a and k_b , which are their private keys and compute $A = k_a * P$ and $B = k_b * P$ respectively. They then exchange the results, A and B , and the shared secret is $k_a * k_b * P = k_a(k_b * P) = k_b(k_a * P)$.

Security

- details here + DH + ECC
- <https://ocw.mit.edu/courses/mathematics/18-704-seminar-in-algebra-and-number-theory-rational-points-on-elliptic-curves-fall-2004/projects/asarina.pdf>[1]

The security relies of this algorithm on elliptic curve discrete logarithm problem. Instead of obtaining the power a from g^a , one needs to find a k , if exists, such that $kP = Q$, where P, Q are points on the elliptic curve over a finite field $F_q, q = p^n, p$ prime.

Usually, the methods for solving this problem are slow, but there are certain types of curves that are vulnerable. - attack methods [2] - baby step, giant step method - general - MOV method - for elliptic curves over finite fields described as F^x_q , when m is small - index calculus method

3. Existing Technologies

- about the technologies used in some of the popular end-to-end encrypted apps and a brief description of how they work

Signal protocol

- Signal protocol - wiki
- Signal docs
- The X3DH Key Agreement Protocol
- A Formal Security Analysis of the Signal Messaging Protocol
- Hecar Lexicon: What Is the Signal Encryption Protocol?
- https://en.wikipedia.org/wiki/Montgomery_curve
- A Formal Security Analysis of the Signal Messaging Protocol - notes on how the protocol works

- some words about OTR, SCIMP, Axolotl ratchet and how the protocol evolved
- Signal is one of the leading end-to-end encryption protocols at the moment. The applications using this protocol are Signal, Whatsapp, Facebook Messenger in Secret chats, Skype in its Private conversations and Wire.
- It was developed by Open Whisper Systems and used Off The Record Messaging (OTR) cryptographic protocol and initially used by TextSecure, the predecessor of Signal. (2013)
- OTR was the first security protocol for instant messaging and after each message round trip?, the users established a new ephemeral Diffie-Hellman shared secret => ratcheting because you couldn't decrypt past messages
- The second version was based on the Axolotl Ratchet and SCIMP encryption scheme (2014)
- A third version redefined the Axolotl Ratchet as Double Ratchet. After more changes, the application was renamed to Signal (2016)
- combines
- double ratchet algo
- prekey bundle
- x3dh handshake
- uses
- Curve25519
- AES-256
- HMAC-SHA256
- 20. Cohn-Gordon2020_Article_AFormalSecurityAnalysisOfTheSi - for security analysis too
- 20. Signal Protocol - Makalah-Kripto-2020-06.pdf*

- ratcheting, forward secrecy
- X3DH
- 3 stages:
 - initial key exchange with X3DH - long term, mid term and ephemeral DH keys to produce a shared secret root value
 - async ratchet - users alternate in sending the new ephemeral keys with prev generated root keys to generate forward secret chaining keys
 - sym ratchet - use key derivation functions to ratchet forward chaining keys to create sym enc keys
- => each message is encrypted with a new message key
- the ping pong pattern of new ephemeral keys inject auto entropy
- TextSecure - used Double ratchete, called Axolotl ratchet at that time
=> RedPhone => Signal
- over 10 diff types of keys and a chain of updated keys
- async transmission protocol which requires pre-send batches of ephemeral public keys
- when a sender wants to send the messages, they get the keys and performs an AKE like protocol using the long term and ephemeral keys to compute the message encryption key
- the message keys depend on previous computations of the keys
- registration - users register their identity w/ a key distribution server and upload the long, mid term and eph keys
- session setup - get the public keys of the recv and establish initial enc keys (x3dh)
- sync messaging (asym ratchet updates) - sender exchanges their public keys with the recv and generate a shared secret => start chains of message keys, fresh ephemeral keys

- async messaging (sym ratchet) - a new sym message key is derived from the previous state, if no new message was sent by the recv, keys derived from the previous ephemeral dh public key of the sender
- the following are present there *
- uses X25519 or X448 ECDH
- key derivation functions: HMAC SHA256, HKDF SHA256
- AEAD - encrypt then MAC scheme: AES256 in CBC, PKCS#5 padding, MAC is HMAC sha 256
- xeddsa signature scheme - x25519 or x448
- the rest is highlighted in the other version of the paper

Diffie Hellamn

- <https://www.cs.jhu.edu/~rubin/courses/sp03/papers/diffie.hellman.pdf>
- Diffie-Helman key exchange (colors)
- Diffie-Helman key exchange (maths)
- How Signal Instant Messaging Protocol Works - more explanations
- How End-to-End encryption Works? - wapp
- Double Ratchet Messaging Encryption
- Key Exchange problems
- Elliptic curve

Extended Triple Diffie Hellamn

- x3dh
- Extended Triple Diffie Hellamn is the key exchange protocol used by Signal and provides forward secrecy and cryptographic deniability
- It was designed for asynchronous communication, so the users need to provide some information to the server such that the others can

establish the secret key without both being online; also, the public keys are used to authenticate the users

- To prepare the setting, the communication will have 3 parties: the users, A (sender) and B (receiver), and the server
- the algorithm used by the application needs the following parameters: a curve (X25519 or X448), a hash function (SHA 256 or 512) and information that identifies the application; it also needs an encoding function to encode the public key into a byte sequence
- the keys are based on the elliptic curve previously chosen as parameter and they are as follows:
- long-term identity key - each party has one and they are public
- ephemeral key pair - generated at each run with the public key
- signed prekeys and a set of one-time prekeys - these are sent to the server so that the other party can establish the key exchange
- the shared secret is a 32 byte secret key
- the protocol has 3 phases:

Key publishing - B publishes his id key and prekeys to the server as follows:

- the id key is uploaded once - the signed prekey and prekey signature are replaced after a time interval - a set of one-time prekeys, which are changed after each run - the private keys of the previous signed prekeys should be deleted as fast as possible after they have decrypted the message, so forward secrecy can be kept

Initial message - A gets the prekey bundle from the server, used to send the first message to B - the prekey bundle should contain the keys stated previously but the one-time prekeys are optional; if the server sends one, it should be then deleted - after the prekey signature verification is successful, A generates an ephemeral key pair and uses it, along with the signed prekey and identity key of B, to generate 3 key pairs that will compose the secret key; if the prekey bundle contains a one-time prekey, another key pair is generated using it and the ephemeral key - after the shared secret is computed, the

ephemeral private key is deleted - from the identity keys of the parties, an associated data sequence is generated, which may have other additional information appended, such as identifying information, certificates, usernames etc. - the initial message contains: - the identity and ephemeral keys of A - information about B's used keys - a ciphertext encrypted with an AEAD scheme which uses the associated data sequence and the encryption key is either the shared secret or the output of a pseudo random function (PRF) keyed by the shared secret

Receiving the initial message - to receive the initial message, B needs to obtain A's identity and ephemeral keys from the message - B then follows the same steps to compute the secret key and creates the AD sequence using the identity keys so he can decrypt the message using it and the shared secret - to keep the forward secrecy, he deletes the one-time prekeys used for this message

- in both cases, the parties may continue to use the same secret key or derivations of it after the exchange

Security considerations - authentication - the parties can compare their id public keys using an authenticated channel (ex QR code scanning, compare pk fingerprints) - if they don't do this, there is no guarantee that the parties are who they claim to be

- replay and key reuse
- if no one-time prekey is provided, the message can be replayed and accepted by the receiver
- this could raise problems because the same secret key can be derived in different runs
 - to avoid
 - * after the key exchange, a new encryption key for the sender can be negotiated from a fresh input from the receiver
 - * blacklist observed messages
 - * replace old signed prekeys earlier
- deniability
- provides no proof that that of the contents or that the communication took place

- if one of the participants is collaborating with a third party, then proof of their communication can be provided
- signatures
 - if there is no signature verification - the server could provide forged prekeys => key compromise
- key compromise
 - compromise of private keys => could lead to impersonation or affect the security of secret key values
 - mitigation (a kind of) - use of ephemeral keys and prekeys
 - some compromise scenarios (pg 9)
- server trust
 - malicious server could cause communication failure, can refuse to deliver messages
 - if the parties are auth, the server might refuse to send the one time prekeys and then the forward secrecy of the secret key depends on the signed prekey's lifetime
 - if one party attempts to drain the one-time prekeys of the other party, the server should prevent this (ex: rate limits on fetching the prekey bundles) - affects the forward secrecy
- identity binding
 - auth doesn't prevent identity misbinding/ unknown key share attacks
 - an attacker could impersonate a party by presenting their prekey bundle as someone else's
 - mitigation: add more identifying info in the associated data, hash more identifying info etc.

Double Ratchet

- double ratchet
- after the key exchange, the parties are using the Double ratchet algorithm to exchange messages
- new keys are derived, combined with results from previous DH computations so that the keys cannot be recalculated later
- this algorithm uses a cryptographic function, KDF, which provides a

seemingly random output from a random secret key and the input data

- this is further expanded into KDF chains, which use as input and output key parts of the output of another KDF
- this type of function then has the following properties, as stated in the paper:
 - resilience - the output keys appear random if the keys are not known
 - forward security - past keys appear random if the current ones are known
 - break-in recovery - future keys appear random if the current ones are known
- each party has 3 chains for each session: root, sending, receiving
- a new concept, called DH ratchet, is described as follows: the newly exchanged DH secrets, during message exchange, become the input to the root chain and the outputs of the KDF are the keys for the sending and receiving chains

Symmetric key ratchet - the chains advance with each message sent and received and their unique output keys are used for the encryption and decryption; this is called a symmetric key ratchet and the unique keys are message keys - they don't provide break-in recovery because the input of the KDF is constant and they can be stored unordered or lost messages can be easily handled

Double Ratchet - to prevent previous and future messages compromise, the protocol combines the symmetric key ratchet and the DH ratchet, resulting in the double ratchet algorithm - a new DH key pair is generated, being the current ratchet key pair; when a new ratchet key pair is received from another party (from the message header), the current one is replaced - when a new message is sent or received, the symmetric key ratchet step is applied on the corresponding chain (sending or receiving) - in this way, if one of the private keys is compromised, it will soon be replaced with another one, providing both forward and backward secrecy

- in case the messages are sent out of order or some are lost, the messages header contains the number in the sending chain and the length in the previous sending chain /x, so the receiver will store the message keys for those messages

- pg 20

EdDSA signature

- xeddsa
- create and verify EdDSA signatures using the key pair defined for X25519 or X448
- this scheme is defined as xeddsa with the extension vxeddsa, which provides a verifiable random function(VRF)
- signatures are defined on twisted Edwards curves and use the SHA 512 hash function
- the input parameters for the signature algorithm are the (Montgomery) private key mod integer q , the message and 64 bytes of secure random data and for the verification algorithm: the (Montgomery) public key, the message and the signature
- the signatures are randomized
- signing time is constant
- it is considered safe to use the same key pair to produce the signatures

Sealed sender

- sealed sender
- available for the Signal app
- this feature provides sender anonymity because the identity of the sender is hidden from the service provider
- the users have a certificate that attests their identity; they are periodically changed and contain the phone number, public identity key and the expiration date; it can be included in the messages too

- the users derive delivery token from the profile key, which is 96 bits long, and register it to the services; in order to send the sealed sender message, the user needs to prove that they know the delivery token for that user
- the profile keys are also encrypted and the profiles are shared between the contacts
- but the Signal app allows the users to receive sealed sender messages from outside their contacts list, but they won't be able to authenticate the senders first
- the messages are encrypted in a normal fashion, but then the sender certificate and the ciphertext are also encrypted

Security analyses

- 16 analysis
- double ratchet started with TextSecure, which was the app developed before Signal and it combined the ideas from OTR's asymmetric ratchet with a symmetric ratchet; this one didn't include parts from the DH exchange, it only derived a new symmetric key; this was referred to as Axolotl ratchet
- unknown key share attack (UKS) - is a type of attack where a communication between two honest users is targeted by an adversary at key exchange; one of the users thinks that they shared key with the recipient, but the recipient is unknowingly sharing the key with the attacker; this attack can be mitigated by including both user identities in the key derivation function
- this type of vulnerability was present in the TextSecure protocol and is not prevented by Signal, since the key derivation is not based on the identities of the users too
- the paper focuses on the "multi stage AKE protocol" part of Signal
- the paper defines the following threat model:

- the network is fully controlled by the adversary
- the authentication is implicit, meaning that the intended party can compute the key $/x$
- side channel attacks are not taken into consideration and the out of band verifications of the long and mid term keys is assumed
- 21 sealed sender - improvements
- according to this paper, the anonymity that this feature offers to provide is broken if more messages are sent
- the attack proposed is of the statistical disclosure attack (sda) type and uses the delivery receipts
- the sealed sender feature hides the identity of the sender from the server or any third party and can be decrypted by the receiver, who will then learn the identity of the sender from the payload
- the identity of the sender is included in the payload to be used later for message authentication
- a delivery token of 96 bit length is derived from the user profile of the sender so the sender proves that they know the receiver -> this is used to prevent spam
- the threat model presented is based on a type of sda, which can be used on message timings, so a link between the sender and the receiver can be made
- this is plausible with the assumption that the response from the receiver is fast; this is facilitated by the usage of delivery receipts
- the delivery receipts are enabled by default and they tell the sender whether the message was only delivered or it was received by the recipient's device
- the attack is based on the fact that the sender appears as a recipient in the time period immediately after the initial receiver got the message
- computations and other details at 6-8

- this concludes that the average number of messages exchanged between the two parties would be under 5 to obtain the identity of the sender
- proposal: sealed sender conversations - the initiator of the conversation is not leaked (3 solutions are briefly presented at page 2)
- this improvement would target the whole conversation between two users, not just the messages
- the identity of the receiver can be visible, but the sender's shouldn't be during the whole lifespan of the conversation

Extra

- Statistical deisclosure control
- technique that ensures that ensurea that no persona or organization is identifiable from the results of an analysis of survey or administrative data or in therelease of microdata
- or protect the confidentiality of the respondets and subjs of the research
- related to the statistical disclosure attacks

Terminology used

- target - bob, user being monitored
- associate - alice, the user sending msg in the attack window to the target
- non-associate - charlie, not sending msg to the target
- attack window - attack time frame, spanning multiple messages to the target
- target epoch - single epoch during the attack window immediately after a sealed sender msg to the receiver
- random epoch - same len as target epoch, but randomly chosen and ind from bob

Whatsapp

- wapp whitepaper

- whatsapp follows the Signal protocol for registration and key exchange; it also implements the key chains used to derive message keys from the double ratchet algorithm
- end-to-end encryption is enabled by default, from 2017
- the attachments are encrypted with AES 256 in cbc mode and a random IV and a MAC of the ciphertext is appended using HMAC SHA256; two 32 byte keys are randomly generated, for AES 256 and a HMAC SHA256 ephemeral one /x
- the attachment is uploaded to a “blob store”? and the encryption key, ephemeral key and the SHA 256 hash of the encrypted blob and a pointer to the blob store /x is sent to the recipient
- the recipient will then obtain the keys, retrieve the encrypted blob, verify the hash and the MAC and then decrypt the message
- to verify the identities of other participants, the users can scan their QR code
- for the transport layer, Noise Protocol framework is used, with curve25519, aes gcm, sha 256

Groups

- for groups, Whatsapp uses a “server-side fan-out” method for sending the encrypted messages to each member; this means that the sender transmits a single message to the server and the server will distribute it to each participant
- this uses pairwise encryption and the Sender Keys component of the Signal protocol /x
- first message:
 - sender generates a 32 byte chain key and a signature key pair, over curve25519
 - the chain key and the public signature key are combined and form a Sender Key message which will be individually encrypted and send

using the pairwise model to each member

- the rest of the messages:
- a new message key is derived from the chain key and it is updated
- the message is encrypted using AES 256 CBC and signed with the signature key
- the sender then transmits the message to the server which forwards it in a fan-out fashion to the participants
- with the chain key, the forward secrecy is maintained and the sender key of each participant is reset when one member leaves

Calling

- video and voice calls are also encrypted
- after a session is initialized, the initiator generates a random 32 byte SRTP master secret $/x$ and sends an encrypted message containing it to the recipient

Analysis - 2016-17 analysis - this analysis paper mentions a lack of a thread model in the whitepaper and they provide one that can break the protocol either by decrypting the ciphertext of more messages or by impersonating a user - the issue found is that after, the session is established, some of the parameters of the same session are used, unless an external event (device change, reinstalling the app) occurs, and without scanning the QR code (the parties verifying each other's identity), MITM attacks are possible so the integrity of the messages is at risk so another verification step is recommended

MTPROTO

- <https://core.telegram.org/mtproto>
- <https://core.telegram.org/mtproto/description>
- <https://core.telegram.org/api/end-to-end/video-calls>
- <https://www.cybercitadel.com/signal-vs-telegram-a-detailed-comparison-of-security-and-privacy/>
- <https://telegram.org/evolution>

General description

- the protocol was created in 2013 for the Telegram messaging app, which was founded by Nikolai and Pavel Durov
- docs
- from version 4.6 (dec 2017), the protocol was upgraded from MTProto 1.0 to MTProto 2.0 but the first version is still supported for backward compatibility
- the MTProto 2.0 uses
 - SHA 256 instead of SHA 1
 - padding
 - the message key depends on the message and a portion of the secret chat key
 - 12 - 1024 padding bytes used instead of 1 - 15
- end-to-end encryption is not enabled by default and this is considered a drawback when compared to other apps
- the end-to-end encrypted chats are called “secret chats”, option which should be explicitly stated by the user, and the group messages are not encrypted at all
- has 3 components:
 - high lvl component (API query language)
 - cryptographic layer
 - transport component
- High lvl component/ API query language
- the session in which the client and the server communicate, which is attached to the application on the client device and a user ID, for authorization purposes
- Authorization and encryption

- before the message is sent, it is encrypted and an external header is added on top of it, with a 64 bit key identifier and a 128 bit key => user key and the message key return a 256 bit key and initialization vector which will be used to encrypt the message with AES 256 in infinite garble extension (IGE) mod /x
- the initial part of the message contains data about the session, message ID, sequence number, server salt
- the message key is the 128 middle bits of SHA 256 of the message body prepended by 32 bytes from the authorization key
- the authorization key is 2048 bit key and is generated at first app run and is not changed (almost never) => no backward secrecy if the device is compromised
- the creation takes place by exchanging Diffie Hellman keys with the server /x??
- how they overcome this:
- session keys with Diffie Hellman key exchange
- protect the key on the device with a password
- encrypt cached data
- if the time is not sync for both the client and the server, the server sends a message with the correct time and a 128 bit salt and some identifiers, which will be checked by the client; if a time correction was not performed, a new session will be generated
- Transport
- the payload/ message is wrapped in a secondary header, defined by the appropriate MTProto transport protocol /x
- a list with them here /4-5
- some terminology and key details, from here
- the message is encrypted and an external header is added on top of it;

it contains a 64bit auth key id and a 128 bit message key, defined as the 128 middle bits from the SHA 256 of the message body prepended by 32 bytes taken from the authorization key /x

- the authorization key with the message form a 256 bit key and initialization vector that will be used to encrypt the message using AES 256 in infinite garble extension (IGE) mode /x
- the initial part of the message contains information about the session, message ID, message length, sequence numbers and server salt
- the authorization key is 2048 bit long and is created on user registration; it is exchanged between the client and the server using the Diffie Hellman protocol; these rarely change
- the server key is also 2048 bit long and is a RSA key used to digitally sign the messages on registration and when the authorization key is created
- the key identifier is used to show which key was used to encrypt a message; it is made up of the first lower-order of the 64 bits of the SHA1 hash of the authorization key; they are unique and if a collision happens, the authorization key is regenerated; this version of the hash algorithm is still used because the authorization key should be identified independently of the protocol /x
- the session is identified by using a randomly generated (by the user) 64 bit number
- server salt is required to protect against replay attacks and time desync with the server /x; it is a random 64 number that is changed periodically; the change is requested by the server and all new messages should contain the new salt
- message identifiers are also 64 bit numbers that uniquely identify the message in a session and must be approx equal to $\text{unixtime} \cdot 2^{32}$ so that the time it was created can be determined
- the message key is the middle 128 bits of the sha 256 hash of the plaintext message and the message details, prepended by the 32 byte

fragment from the authorization key; compared to MTProto 1.0, the last 128 bits of SHA 1 hash were taken into account and no padding and authorization key fragments were involved

- internal cryptographic header - server salt and the session => 16 bytes header added before the message is encrypted
- external cryptographic header - 24 bytes header added before the message encryption and it has the authentication key identifier and the message key
- the AES 256 key and the 256 bit initialization vector are computed from the authorization key and the message key

Secret chats - section from the docs

- the key generation and exchange are done using the Diffie Hellman protocol
- the initiator of the request needs to obtain the prime p and generator from the server and check if p is 2048 bit number and is it a safe prime
- private keys used are also 2048 bit long
- the shared secret key's fingerprint, which is used for detecting bugs, is equal to the last 64 bits of the SHA 1 hash of the key
- forward secrecy is kept by re-initiating the keys after it was used to encrypt or decrypt 100 messages or after a week, the old keys being discarded
- files are encrypted using one-time keys that are not related to the shared key
- the AES key and the initialization vector are two random 256 numbers
- it is assumed that the address of the file is attached to the outside of the encrypted message using a file parameter and the key is sent in the body of the parameter /x

Security analyses

- symbolic verification
- code for the client and the protocol itself is open source, but not the servers
- there are not enough security analyses on this protocol and was criticised for the choice of cryptographic primitives and the encryption mode

Previous issues found - these issues were found in version 1.0 - the padding was added after the encryption, so this could lead to chosen ciphertext attacks - no forward secrecy was provided - the message sequence numbers were managed by the server, so there was place for replay attacks - the server could perform a MITM attack on two parties because the Diffie Hellman exchange protocol was used before the participants were authenticated; the first 128 bits of the hash, using SHA 1, were used for the fingerprint and the server could perform a birthday attack to find the keys corresponding to this fingerprint; this was changed in 2.0 and the fingerprint is 288 bits long, containing parts from the SHA 256 key /x

- the current version is considered to guarantee integrity of ciphertext, indistinguishability of chosen ciphertext and forward secrecy, as no attacks of these types are known

Study findings - the study conducted finds that there are no logical flaws in the protocol, and possible vulnerabilities can arise from the cryptographic primitives used, implementation flaws (not enough checks), side-channel exfiltration (time and traffic analysis) /x or incorrect user behaviour /x - also, the messages are kept secret after the re-keying process is completed, even if the authorization keys are compromised for both participants before the process - forward secrecy is also kept but if the attacker recovers the session key, the last 100 messages or the messages in the past week are compromised - if the parties have correctly validated the initial session keys, then the messages they exchange are authentic; otherwise, MITM attacks are possible, so the participants need to be aware of the correct way in which they perform the procedure - the server can send out weakened Diffie Hellman parameters such that the discrete logarithms become easier to compute, so the end user need to verify these parameters

- security analysis on mtproto 1.0
- SOME OF THESE THINGS ARE DEPRECATED
- the whole study shows that you could use the metadata to “guess” when the communication between two parties takes place
- is considered to provide the best user experience because of its speed and functionality
- the reason they gave for the fact that end-to-end encryption is not enabled by default is that the secret chats are bound to the device and that it is not possible to continue a conversation on devices where this feature is not enabled
- the official clients are open source but have binaries without public codes
- Telegram also provided a CLI lol
- cloud storage, so the server has access to all unencrypted messages and metadata
- the paper classifies the issues as follows:
- non-technical concerns
 - end-to-end encryption is not by default
 - the protocol is “home grown”
 - access to the contact list needs to be provided beforehand and the information is stored on the server
- technical
 - MITM attacks (2015), by generating Diffie Hellman secrets that provide the same authorization fingerprints with the same 128 bits as the ones of the users, so they can’t detect the attack
 - used a modified version of the Diffie Hellman protocol until 2014, where the key was XORed with a nonce, so the attacker could use different nonces and the users would have the same key which would be also known to the server /x

- SHA 1 instead of SHA 256, which is not collision resistant
 - third parties could observe metadata about the users - an example of this is the availability leak, meaning that the observer (someone who has their phone number) could see when the communication between two users might take place
- cca
- study on the fact that MTPProto is not IND CCA secure and doesn't satisfy the authenticated encryption definitions, but this was fixed in version 2.0
- the attack is only theoretical
- the protocol uses a block cipher structure, so it has padding, but it is not checked for integrity, so one can create two different ciphertexts that decrypt to the same plaintext
- the two attacks presented are
- padding length extension, which means adding a random block at the end of the ciphertext; the padding is not checked by the authentication function; to mitigate this type of attacks, you need to check the length of the padding during decryption
- last block substitution, or, in other words, replacing the last block with a random block; mitigation by adding the padding when computing the authentication tag, but this breaks backward compatibility
- authentication problem
- this is referenced <https://www.cryptofails.com/post/70546720222/telegrams-cryptanalysis-contest>
- the possibility of a MITM attack is presented here aswell, with the fingerprints, but the attacker needs to make the victims install a malicious client or modify the installed one
- also the use of IGE was criticised

Signcryption

- Official
- E2ee official
- Chosen Ciphertext Attacks on Apple iMessage
- more on signcryption (definitions) - 2002?
- this protocol is used by iMessage, with which the widespread adoption of secure instant messaging protocols started

Apple iMessage - official docs (this section) say that iMessage and FaceTime use end to end encryption, attachments are also encrypted (are uploaded and deleted after 30 days if the message was not sent) - you can choose automatic deletion of the messages - info stored: - use of services - messages that can't be delivered - held for 30 days - metadata about FaceTime calls, 30 days - contacts, 30 days - encryption is enabled by default

Apple security and encryption - Encryption and data protection, official (this section) - Security overview - APN - Apple Push Notification service - IDS - Apple Identity service - encryption - RSA 1280 bit key, EC 256 bit key on NIST P 256 curve - signatures - ECSDA - private keys are saved in the device's keychain (API for passwords, keys and other sensitive credentials) - public keys are sent to IDS

- How iMessage sends and receives messages securely
- methodology - Data Protection
- sender retrieves the public keys and APNs addresses for all associated devices of the receiver
- the message is individually encrypted for each device
- the sender generates a random 88 bit value that is used as a HMAC SHA 256 key to create a 40 bit value, from the sender and receiver's public keys and the plaintext; this value will be used to verify the integrity of the message, after decryption

- $88 + 40 = 128$ bit key, which encrypts the message using AES in CTR mode
- AES key is encrypted using RSA-OAEP public keys of the recv device, but iOS 13 or later might use ECIES encryption
- the messages then contain the encrypted message, encrypted message key and the sender's digital signature; metadata (timestamp, APN routing info) is not encrypted
- the combination of the encrypted message and the encrypted key are hashed using SHA 1 and signed with ECDSA
- if message too long or has an attachment, it is encrypted using AES in CTR mode with a randomly generated 256 bit key and uploaded to the cloud; the receiver gets the AES key, the URI and a SHA 1 hash of the encrypted form?, encrypted
- this process repeated for each participant in case of group communications

Signcryption

- Wiki
- introduced in 1997, along with an elliptic curve signcryption (saves 58% computational and 40% communication costs, compared to the classical signature-then-encryption schemes)
- public key
- functions of both digital signature and encryption, so you don't digitally sign the message and then encrypt it and in this way you decrease the cost and optimize the procedure
- key generation, signcryption, unsigncryption
- properties:
- correctness

- efficiency
- security, and some signcryption schemes provide public verifiability and forward secrecy, so:
 - confidentiality
 - unforgeability
 - non-repudiation
 - integrity
 - public verifiability
 - forward secrecy and message confidentiality
- Zheng
- this primitive was first introduced to reduce the cost of the signature and encryption operations, by combining them
- the total cost can be considered the sum of costs of each operation and using this combined approach would reduce it with, as specified in the paper, 50% in computational cost and 85% in communication overhead, keeping the security definitions offered by the two operations, namely sending an authenticated and secure message
- the primitive would contain a pair of signcryption and unsigncryption algorithms that are satisfying the conditions:
- unique unsigncryptability - the ciphertext run through the unsigncryption algorithm would return the message unambiguously $/x$
- security - the algorithms fulfill the properties of a secure encryption scheme and secure digital signature: confidentiality, unforgeability, non-repudiation, integrity
- efficiency - smaller cost on applying the scheme
- this paper explores a version based on shortened ElGamal signatures and encryption
- security of signcryption
- two definitions, depending on the adversary: outsider or insider

- signcryption is defined for the public key setting, being the equivalent of authenticated encryption in the symmetric encryption
- the paper explores the applicability of this scheme in a setting with two users, but this can be generalized to multiple users
- the scheme needs to fulfill the following conditions?:
- each user should publish their public key
- the signcryption needs to contain the identities of the sender and the receiver
- each user should be protected from the other users $/x$
- the scheme has three algorithms:
- generation, which returns the pair of keys: the private sign and decrypt key and the public verification key
- (randomized) signcryption, which takes as parameters the sender's secret key, the receiver's public key and the message $/x$
- (deterministic) designdecryption or unsigncryption, takes as parameters the ciphertext, the receiver's secret key and the sender's public key

Security - pg 8 - outsider security - the adv knows the public keys and has access to the signcryption and unsigncryption oracles of the sender and the receiver, so it can do CPAs and CCAs; the scheme is outsider secure (indistinguishable against generalized CCA2) if it is outsider secure against adaptive chosen ciphertext attacks; OR it protects the privacy of the receiver when talking to the sender from outsider intruders who don't know the secret key of the sender - insider security - the adversary is a user of the system so the sender has access to the secret key of the receiver and for encryption, the receiver has access to the decryption key because it contains the secret key of the sender; the scheme is insider secure against gcca2 or cma attacks on the privacy or authenticity property if the corresponding induced enc/sign scheme is secure against the attack (IND gCCA2 - enc, UF-CMA - sign); OR one of the parties is the attacker, so the scheme protects the sender's authenticity against the receiver and the receiver's privacy against the sender

- non-repudiation - signcryption only assures the receiver that the message was sent by the actual sender
- comparison btw insider and outsider security:
- is for authenticity implies non-repudiation when the receiver reveals its secret decryption key or by zero-knowledge proof
- os - the receiver can generate valid signcryptions of messages that are not actually sent by the sender, so non-repudiation can't be achieved
- os might be enough for privacy and authenticity, but the is can be useful in case the secret key of the sender is stolen and we don't want the attacker to be able to understand previously or future recorded signcryption (some kind of forward and backward secrecy)

Definitions and notations - EXTRA - indistinguishability - given a randomly selected public key, not probabilistic polynomial time (ppt) adv can distinguish between the encryptions of two chosen messages (by the adversary) - based on the def above there are two stages - find and guess - CPA - chosen plaintext attack - the adv is not given other capabilities but to encrypt the messages with the encryption key - CCA1 - lunch time attack, give the adv access to the decryption oracle and they can decrypt chosen ciphertexts - CCA2 - adaptive chosen ciphertext attack - access to the decryption oracles in the guess stage (def above) and has the restriction that the given ciphertext is different from the actual ciphertext the adv wants to decrypt - secure against gCCA2 - generalized CCA2 security and the scheme is secure against this if there is an efficient decryption respecting relation $R(e, e') \Rightarrow \text{dec}(e) = \text{dec}(e')$

- UF - existential unforgeability - negligible probability of generating a valid signature for a new message
- NMA - no message attack - adversary has the verification key (public)
- CMA - chosen message attack - adversary has access to the signing oracle and query it to obtain valid signatures
- combined: UF-NMA, UF-CMA
- sUF - strong unforgeability - adv should NOT be able to generate a signature for a new message and to generate a diff signature for a signed message -> makes sense for CMA, so you have sUF-CMA

Security analyses

- Security under Message-Derived Keys Signcryption in iMessage
- iMessage uses signcryption to encrypt the messages
- iMessage uses a scheme that involves symmetric encryption with the key derived from the message
- EMDK (encryption under message derived keys) is the scheme (as named in the article) that is the primitive used by this protocol
- the protocol was vulnerable to chosen ciphertext attacks, CVE 2016 1788, and it was revised later
- EXTRA - as is pointed out in the article, Msg1 is the version before the revision and Msg2 is the one after
- the aim of signcryption is to provide privacy and authenticity of the message, so it can be perceived as the “asymmetric analogue of the symmetric authenticated encryption” /x
- EXTRA
- a formal definition was given by ADR (will be addressed) and they distinguish between outsider security - the adversary is not one of the users - and insider security - adversary is one of the users
- the Msg1 scheme can be considered a simple (ADR) encrypt then sign method and they provide an attack vector, showing that it is not IND CCA (not insider secure); the Msg2 version protects against this type of attacks
- the algorithm is multi-recipient, meaning that one can send messages to a receiver and they can access them on every device
- as defined, the EMDK scheme takes in the message and returns a key and the ciphertext and to decrypt, the algorithm uses the key and the ciphertext; the key is sent using asymmetric encryption
- two security requirements are defined: an adapted authenticated encryp-

tion requirement for the symmetric encryption and a form of robustness, wrong key detection method /x

- cca
- the analysis shows that practical adaptive chosen ciphertext vulnerabilities are present and that the attacker can retrospectively decrypt ciphertext (payloads and attachments) in a relatively short time (2^{18} queries)
- this type of attack operates on gzip compressed data and they call it “gzip format oracle attack”
- about the conducted attacks: they provide attacks that are retrospective, meaning that the attacker needs one of the target devices to be online and has access to the ciphertexts
- some limitations of the protocol
- key server and registration: the server (IDS) is operated by Apple, so if it is compromised, it can become the platform for MITM attacks /x; they don’t provide a way for the users to verify the authenticity of the keys
- no forward secrecy: the encryption keys are rarely changed and the protocol doesn’t provide any forward secrecy mechanism
- replay and reflection attacks: no mechanism that prevents replay or reflection attacks, so the attacker can falsify conversations or, if they have access to the device, replay previously captured traffic and obtain the plaintext /x
- no certificate pinning on older versions: devices running older versions of iOS are still vulnerable to MITM attacks because of the lack of certificate pinning
- seemingly ad-hoc cryptographic scheme: combines RSA, AES and ECDSA
- and attacks on the encryption mechanism, in two stages:

- the encryption mechanism has weaknesses because, instead of using a MAC or AEAD modulus operandi, it uses ECDSA to guarantee the authenticity /x of the ciphertext (symmetrically encrypted portion of the message payload /x), which is not sufficient because the attacker can change the signature from an account controlled by the them /x
- the attacker has the ability to modify the AES ciphertext and then can recover the plaintext from the decryption provided by the target device (CCA2)
- these made the attack possible
- gzip, version of DEFLATE compression, combines LZ77 and Huffman coding to compress common data types
- the attacker intercepts the gzip compressed message encrypted with an unauth stream cipher and has access to the decryption oracle,
- EXTRA: some references to the fact that the some states (USA mostly) wanted to have backdoors into the end-to-end encryption schemes of the app
- 2020 paper

Letter Sealing

- <https://linecorp.com/en/security/encryption/2020h1>
- <https://help.line.me/line/?contentId=50001520>
- the app that implements this protocol is LINE
- introduced in 2015, default from 2016 on new versions
- whitepaper
- <https://d.line-scdn.net/stf/linecorp/en/csr/line-encryption-whitepaper-ver2.0.pdf>
- transport protocol: based on SPDY

- handshake protocol based on 0-RTT
- the transport protocol uses secp256k1 curve for key exchange and server identity verification
- symmetric encryption - AES + HKDF for key derivation
- static ECC key pairs and the private part is stored on the server and the public one on the device? (for TLS)
- the key exchange is done using ECDH and the server identity verification with ECDSA
- handshake protocol, you need to exchange some data first:
- client:
 - generate ephemeral ECDH key + 16 byte client nonce
 - derive temp transport key and initialization vector (16 bytes long) using the server's static key and the ephemeral key generated before
 - ephemeral ECDH client handshake generated
 - gen ephemeral ECDH client handshake key
 - encrypt the application data with the ephemeral key and initialization vector and send the static key, the public key, the nonce and the encrypted data to the server
- server:
 - temp transport key and initialization vector using server's static ECDH key and client's initial eph key
 - decrypt recv app data and get public key
 - generate eph key pair and nonce - 16 bytes
 - derive forward sec transport key and init vect
 - gen sign and handshake state using server's static key
 - encrypt app data
 - send to client: server public key, server nonce, server's static signing key, enc data
- client finish:

- handshake signature verif
- if verified, get the forward secrecy key and initialization vector
- enc using the key and the initialization vector
- application data is encrypted with 128 bit key using AES GCM AEAD cipher
- the server and the client generate a nonce, computed from a 64bit long sequence number and the initialization vector from the handshake
- the encryption function takes as parameters the key, nonce and application data?

Message encryption - only messages and metadata are encrypted - the protocol was updated to version 2 (after vulnerabilities were reported in 2018, discussed later) and it also adds integrity of the message metadata - version 2 is enabled by default unless one of the participants doesn't support it; in that case, the message is sent again using version 1 - the primitives used (comparison between versions with /) are: (this is a table) - for key exchange: ECDH over curve25519 - message encryption, data authentication: AES 256 in CBC/ GCM - data authentication: AES-ECB withSHA256 MAC/ AES 256 in GCM - SHA 256/ no message hash function? - message data: encryption and integrity? - it refers to the fact that the mode is AEAD - message metadata: not protected/ integrity?

- message encryption protocols are different for each type of the provided environments (presented below)
- private messaging
- the key pair is generated at app launch or after the app is reinstalled and it is saved locally
- the public key is registered on the server and it binds the user (using a key ID) to that public key
- the shared secret is established using ECDH and the key can be verified
- message encryption depends on the version supported by the user:
- version 1: using the key and the initialization vector, which are derived

from the shared secret and a random 8 byte salt, with the message as parameters for the AES in CBC mode; a MAC is computed and the following data is sent to the receiver: version, content type, salt, ciphertext, MAC, sender ID, recipient ID; if the key pair is obsolete, the messages are requested to be resent; to decrypt, the receiver needs the init vector, key, the MAC of the ciphertext is compared with the value included and then it is decrypted

- version 2: key and random nonce derived from the shared secret and a 16 byte salt; the nonce is a 8byte per-chat counter with a 4 byte randomly generated value /x; the key and the nonce are used to encrypt the message using AES GCM; message data is encrypted and authenticated and the metadata (sender ID, receiver ID, version, content type) is added associated data; the authentication tag is 16 bytes long and is concatenated to the ciphertext; the receiver gets the decryption key and from the shared secret and the salt, decrypts the message and checks if the tag of the message matches the one computed
- group messaging
- the application generates a shared group key and the private keys for each participant, using their public keys
- then calculates the symmetric keys from the current user's private key (the one who created the group?) and the public keys of the participants
- the key derivation process is similar to the one for private chats
- the shared key is encrypted and sent to the participants and the encrypted data is sent to the server, where it associates the group keys with the group and returns the shared key ID ????
- when something changes between the group participants, a new shared key is generated
- message encryption:
- version 1: the members need to obtain the shared group key and decrypt it; the encryption key and initialization vector are derived from the shared key and the public key of the sender; the rest is similar with the

version 1 private chats, but the key ID of the receiver is changed with the shared key of the group

- version 2: after the shared group key is obtained, the encryption key is composed of the shared group key and the public key; the rest is similar with the version 2 private chats
- for VOIP, the key exchange is done with ECDH over secp256r1
- the caller and callee exchange pairs of ephemeral keys that will be used to generate a master secret $/x$ and derive a VOIP session key and salt, along with a randomly generated unique call ID
- the streams are encrypted with AES CM 128 HMAC SHA1 80 suite
- LINE enc
- provides end-to-end encryption when all of the participants have the Letter Sealing enabled in the following environments
- private chats
- 1-to-n chats
- group chats
- this mode is enabled by default since 2016
- the protocol ensures privacy for the following
- text messages and location in the previously specified environments
- private video and audio calls
- there are a few cases when end-to-end encryption is broken:
- on website preview - when a user sends an URL, the page can be previewed in the chat room
- on spam reports - the messages are sent to the server
- stickers
- open chat?

- group calls
- LINE meetings, social plugin?
- these are encrypted only using TLS - with LEGY*1 (Line Event-delivery Gateway, a custom built API) or HTTPS
- forward secrecy is not by default and few channels support it, such as:
- client-server communication
- in case of key compromise

Security analyses - change from version 1 to version 2 - the papers shows the following vulnerabilities: - impersonation and forgery on group message encryption - a group member could derive the encryption key of another member without knowing the secret key of said member, so this could lead to impersonation; also, this attacker (which could be the server itself) can forge the messages if they bypass the client-to-server encryption - malicious key exchange - a malicious user A can establish a malicious session with a user B in which the shared secret between them is the same as the one used between B and another user C, so A can impersonate both B and C in one of the following ways - message from A to B is sent to C as if it was from B - message from B to C is sent to B as it was sent from A - forgery on authenticated encryption scheme - the attacker can forge a message and that will be considered valid (AES 256 and SHA 256 are combined in a non-standard way)

- some definitions of the attacker are given:
- e2e adversary - bypasses the client-server encryption, so it has access to the messaging server, can intercept, read, modify the messages etc.
- malicious user - a legitimate user of the end-to-end encryption protocol that wants to manipulate the protocol maliciously
- malicious group member - legitimate member of a group with a shared key that want too to manipulate the protocol
- if a malicious member has the shared group key, they can decrypt the messages

Impersonation and forgery attacks - these attacks target a vulnerability in the key derivation phase, because the decryption key and initialization vector are derived from the group key and the sender's public information, so a malicious user can compute any other member's group key and the IV for the message

- impersonation attack
- take the public key of the victim, derive the key and the iv, generate the ciphertext and the message tag and choose the associated data such that it contains the victim's ID and then broadcast it to the group (the attacker can omit the victim) => the message encryption algorithm doesn't provide the authenticity of the message
- forgery attack
- intercept a packet from the victim, obtain the keys and salt from it and the server, decrypt and modify the message, generate a new ciphertext and tag and broadcast the new packet to all members except the victim, to which the original one is sent
- this type of attack can be placed by an e2e adversary, since they can bypass the client-to-server transport encryption
- mitigation: Sender keys used by the Signal protocol, pairwise key exchange

Malicious key exchange on private message encryption - the problem is that there is no key confirmation during the exchange phase and the integrity of the associated data in a packet is not guaranteed (sender ID, recipient ID) - the authentication tag only takes in the ciphertext and the associated data is concatenated with the ciphertext and the tag

- the presented attack is an unknown key share attack - the shared secret key is the same as one used in another session
- if there already is a session between two users, A and B, and one of them trusts a malicious user, C, then C can attack the session between A and B, by establishing a fake session using the same key and IV as the one in that session

- this can be done by registering the victim's public key as C's own public key on the server and to request a new session with the other party; C can now impersonate both participants
- attack 1: impersonate B - send a message to A, which was initially sent from B to C, by impersonating B - can be done by a normal user
- attack 2: impersonate A - send a message (modified) to B, originally from A, as a message from C ? - can be done if the attacker is an e2e adversary
- mitigation: add a key confirmation phase, guarantee the integrity of the associated data by adding them to the hashing function SHA 256, when computing the authentication tag

Security of the Message encryption scheme - it is not a standard one - the adversary doesn't need to know the be trusted by the victims - encrypts a message with AES 256 CBC mode and then tag, computer using SHA 256 on the ciphertext, is encrypted with AES 256 EBC, making the tag computable without any secret info /x - also, in the CBC encryption, the same encryption key is used for the encryption and MAC - forgery attacks, this one consisting of an online and an offline phase; the idea is to compute the tag during the offline phase and get the data in the online phase (left around page 15)

- replay attacks
- replay attack and lack of forwards secrecy, this assumes that the adversary has access to the server
- for the replay attack, the adversary is the server and for the forward secrecy, they require that the private keys of the user are compromised
- replay attack - the adversary can record messages and resend them later to one of the parties and seem to be sent legitimately - the adv can just replace the body of the message
- to protect against replay attacks, one can use MACs which authenticate associated data
- LINE only authenticates the message itself, not the associated data, so the replay attack is possible
- another problems are that the same key is used for both the MAC and the message encryption and that sections, such as the encrypted

- message, MAC and salt could be obtained through reverse engineering
- forward secrecy - it is only provided from client to server and for this attack the private keys of one of the parties is enough

Threema

- open source
- links
- 2020 audit
- the thing used for encryption
- 2021 whitepaper
- the applications are open source
- uses 2 diff encryption layers
- end-to-end
- transport layer, between the client and server
- as cryptographic algorithms, the protocol uses:
- ECDH over curve 25519 for key exchange - asymm keys are 256 bits long
- XSalsa 20 for symmetric encryption - symm keys are derived from the private key of the sender and public key of the receiver + 192 bit nonce -> 256 bits long, as well as the random sym keys used for attachment encryption
- Poly1305 AES for authentication and integrity - MACs are 128 bits long
- forward secrecy is provided on the transport layer
- random padding (PKCS#7) is added before the messages are encrypted
- the protocol offers repudiability

- to avoid replay attacks, the nonces are saved
- local data (caches, contact lists) is encrypted and stored
- the app allows users to sync their contacts list, and it is sent to the server after each email address or phone number was hashed with HMAC SHA 256; because the hashes of the phone numbers can be cracked using brute force attacks, they are discarded after the ID was obtained

End to end encryption

- the messages and attachments are end-to-end encrypted
- key generation:
 - the keys are generated using Curve25519 (the private key includes entropy from the user) and the public key is sent to the server, where the new key is assigned with a Threema ID
 - to retrieve the public key, a user needs to query for the Threema ID, or hash of an email or phone number linked to the ID
 - this system allows more verification levels, the lowest one being the public key only, while the highest is with the phone number
 - additionally, the users can scan a QR code to personally verify the contacts
 - key fingerprints can be created to compare public keys; this is done by hashing the public key with SHA 256 and taking the first 16 bytes
- message encryption
 - the encryption and authentication algorithm uses NaCL library
 - the shared secret is exchanged using ECDH over Curve25519 and hashed with HSalsa20, then the sender uses the XSalsa20 stream cipher with the shared secret and a random nonce to encrypt the message
 - the MAC is computed using Poly1305, where the MAC key is a part of the XSalsa20 key, and it is prepended to the ciphertext
- groups

- in case of groups, the server doesn't keep track of the group structures and the messages are individually encrypted and sent to each member
- the app also provides key backup (pg 9)

Client-server protocol - 3 servers are used

- chat protocol
- used to transfer the messages
- uses a custom protocol with the NaCl library and provides:
 - forward secrecy
 - optimization for minimal overhead /x
 - user authentication with the public key on connection setup /x
- directory access
- used for user signup (creating the Threema ID, linking the email or the phone numbers), getting the public keys
- it uses TLS
- media access
- used to temporarily store attachments
- this data is encrypted with 256 bit symmetric key using Xsalsa20 and has an additional Poly1305 authenticator and uploaded to the server, where an ID is assigned
- the id is returned to the user who then sends the id and the symmetric key to the recipient
- the receiver uses these to download and decrypt the attachment and when this is done, the server is signaled to delete the file
- in case of groups, the data is kept on the server and is deleted after 14 days

Web clients - in case of web clients, the apps communicate between platforms using WebRTC, the packets are end-to-end encrypted using NaCl the signaling with SaltyRTC protocol

VOIP - audio and video calls are also encrypted

Threema Safe - this is for backup

- From the official faq page
- app remote protocol for app and web client data exchange
<https://threema-ch.github.io/app-remote-protocol/>
- Swiss, 2012
- Elliptic curve based (255 bits)
- ECDH with curve25519
- hash function + random nonce = 256 symmetric key for each message
- to encrypt: XSalsa20
- 128 bit MAC added to detect forgeries
- forward secrecy on the network connection
- doesn't require phone number/ email and it provides a Threema ID and contacts access is not mandatory
- personal info is hashed
- contacts or group chats are stored in a decentralized way on the users' devices, not on a server
- messages and media are e2ee
- data stored on servers:
 - messages and group chats until they are sent
 - email addresses and contacts are hashed until the comparizon is done (I guess it's the fact that the other one accepts your request/ you are in their contacts list?)
 - key pairs are locally generated

- no logs on who talks to whom
- for android - AES 256 based encryption for stored messages, media and the ID's private key
- for ios - ios data protection for local encryption
- ID verification to avoid MITM attacks
- Whitepaper
- 3 verif lvls
 - typed id and contact not found in the address book by phone nr or email
 - id matched with one of the contacts
 - persoanlly verified id (scanned the QR code)
- NaCl encryption box model used to encrypt and auth the messages
- SHA256 of the raw public key => first 16 bits are the fingerprint
- to encrypt and decrypt a message
 - ECDH over Curve25519 hased with the result of HSalsa20 => shared secret
 - random nonce generated
 - XSalsa20 stream cipher with the shared secret and the random nonce to encrypt the plaintext
 - sender uses Poly1305 to compute a MAC and adds it to the ciphertext (prepend) a portion from XSalsa20 is used ot form the MAC key
 - sends MAC, cipertext and nonce to recv
 - decrypt and verify authenticity by reversing the steps

Security analyses

- security autdit 2019
- the security audit found mostly minor vulnerabilities or informational and two medium risk ones:

- these are for the Android app, files containing sensitive info (such as key.dat, which contains the private keys of the user) could be sent to another user
- logs with parts of the password for the backup feature, Threema Safe, were saved and could be accessed by an adversary
- some of the low risk vulnerabilities found are:
 - the screenshot saver feature could let you see, for example, the messages in the last open chat, if it was open on application exit, even if the app is protected with a PIN
 - messages could be sent using google assistant, even if the app is locked with a pin
 - for the iOS, there was missing public key pinning in the HTTPS request, which could allow MITM attacks
 - also, an overview of the permissions asked by the app is given and it is stated that these are either necessary or optional; hence, permissions such as access to the voice recording, phone state (for incoming calls) are requested on demand
- Security audit 2020
 - the security audit on the mobile applications for Android and iOS, conducted by cure53 in 2020, shows no high security vulnerabilities, only minor or general flaws which are more related to the devices than to the application itself
 - such issues include the risks of using rooted/ jailbroken devices, since the user can escalate privileges and access files storing private keys and other data or, for Android, that the key used for encrypting and decrypting local data was stored in a file that can be accessed by an attacker with higher privileges if no passphrase is set
 - there is also a screenshot saving feature when the user exists the app which was labeled as a vulnerability with medium security, since sensitive data (such as the messages from the last viewed chat, if it was still open

when the user exit the app) can be exposed when a screenshot of the app is taken (this is still present from the last audit)

Group messaging

- 2020 - Anonymous Asynchronous Ratchet Tree Protocol for Group Messaging
- info is available in this blog post too
- compared to one-to-one chats, group messaging becomes more complex and needs to keep track of all information about the users, such as their keys
- group messaging protocols are implemented in a different way by each previously mentioned application or they are not supported at all
- Telegram, for example, doesn't have ee2e for group chats, for security reasons (here we will have some stories about this. I saw this in the whitepaper or another paper)
- it relies only on TLS and the user needs to trust the server for not using the contents of the messages, as well as Facebook messenger, the content of the messages being available to the server
- if you want to have e2ee in group conversations, you may want to keep forward secrecy, post-compromise security, deniability (if possible)
- bigger groups => gets harder to manage the keys (it is hard in the first place, since you need to do certain things for each participant)
- there are 3 ways in which you can handle this:
- pair-wise - the sender takes the secret key of each of the receiver, encrypts the message and sends it forward to the intended recipient (behaves like normal one to one chats); all the properties of the simple chats are preserved and the groups could be practically invisible for the server

- encrypted message keys - the sender should choose a new random key for each message with which they would encrypt the message and send only a copy to the recipients. The encrypted message keys are then send out to each of the recipients, in order to decrypt the message; the properties are still kept, but the server is aware of the fact that there is a group
- shared group-keys - a group key-exchange should be made (both static and ephemeral public keys from each member); while this would significantly lower the complexity of sending messages, the key exchange complexity could grow, especially when you want to keep forward secrecy and post-compromise security, as the keys must be changed regularly; the server, again, is aware of the group structure
- metadata could be collected, again
- one way to obstruct metadada collection by third parties is to use TLS to secure the comm between the user and client
- the server still needs to know where to send a message
- inside the app:
 - if you can use only email or an username, yo could gain anonimity, but the app would still be aware of your social graph (you will need to find and communicate with the other users anyway)
 - if you have contacts list - use a hash, but this is still not enough, because someone can obtain the tuples of username and hash
- **SGX**
 - wiki
 - some more on this
 - intel
 - set of security related instruction codes that are built into some intel cpus
 - have private regions (enclaves) whose contents are protected and unable to be read or saved by any process (including those with higher priv) outside the enclave

- encryption of some parts of the memory, actually

App comparison - signal - double ratchet procedure for authenticated key-exchange: long term + short term keys from each participant => shared key - key used in a key deriv ratchet protocol => ephemeral session keys - forward secrecy and post-compromise security achieved - ake takes long term enc keys => deniability - initiator sends, in a pair-wise fashion, a fresh keys to each participant - group session key is derived from the shared group key, using double ratchet and the group keys are updated on each newly added participant - online-offline key exchange - static keys are given by the server, but the ephemeral key exchanges are dealt with by having all users frequently upload one-time prekeys to the server, so the inline party can perform the key exchange - safety numbers or qr code for auth - SGX - metadata is stored encrypted on the server

- wapp
- similar, using pair-wise channels
- but the server has access to the metadata
- imessage
- public/ private key pair for encryption and signature, and the public keys are saved on the server
- only static public keys are used => no forward secrecy, post-compromise security
- pair-wise and inefficient, but groups are not used that much, since it was initially created to replace sms
- no way to auth the users
- wire
- double ratchet for session keys for pair-wise comm and channels for group conversations
- auth with safety numbers
- threema

- public keys published to the server and these are used for encr and sign
=> no forward secrecy, post-compromise security
- shared secret keys offer deniability
- group communication with pair-wise channels, but multimedia is encrypted using the message key method
- may offer anonymity - username, phone number or email to create an account
- auth with safety numbers
- 2020 - Challenges in E2E Encrypted Group Messaging
- gives definitions about group anonymity features, such as internal group anonymity and external group anonymity and propose the Anonymous Async Ratchet Tree protocol
- the properties of forward secrecy and post-compromise security should be kept

About MLS?

- CONCLUSIONS SECTION - 2020 - Anonymous Asynchronous Ratchet Tree Protocol for Group Messaging
- messaging layer security - initiative from The Internet Engineering Task Force - protocol for group messaging with the best security properties and more efficient key exchanges (log communication cost, using bin trees)
- MLS
- draft?
- all drafts

Async ratcheting trees - allow async communication where none of the parties need to be online at the same time - keeps backward secrecy - the idea is to generate a group key, encrypt the message once and send it to the group

participants (maybe in a server-side fan-out fashion) - a binary tree is created, where the member device is a leaf node $\Rightarrow O(\log(N))$, N nr of participants - the parent nodes are generated using Diffie Hellman operations between the two children and the intermediate nodes are subgroups \Rightarrow each device might be a member of $\log(N)$ subgroups - the ones in the subgroups can use the key pair to encrypt and decrypt the messages - when a user changes their key or a new user is added, then a new leaf is added to the tree and the updates go up the tree \Rightarrow the root key is changed on each update - this means that all the nodes know the private key of the parent, but not of the other nodes - a node is removed \Rightarrow the path from the root to the removed leaf is blanked and a new shared group secret is computed

TreeKEM - developed based on ART ideas - users arranged in left-balanced trees - the parent key is computed by hashing the key of the last modified node - the nodes know the private key of the parent node, then the updater needs to encrypt $O(\log(N))$ messages and each node receives a message with a secret ++ add and remove operations

MLS

- WIP but would support 2 - 50.000 users
- initially based on ART, but the current versions (as of 2020) are based on TreeKEM
- a few initial implementations available (pg 11)
- hash function for parent nodes replaced with a key deriv function
- the messages are encrypted once and broadcasted as the group operations
- it is assumed that participant has the public keys of all the nodes in the tree, and the private keys for those in the subgroup/ copath (siblings to each node in the path)
- tree verification by recursive hashing
- hash value of each node is based on the info about the current node (leaves) or the hashes of the children (non-leaves)
- running transcript hash value - created using the group ops leading to the current state - each step is a combination of the prev transcript hash function and the current op

- keys and nonces are updated every time the state changes, using a number of key schedules
- the server should provide
- authentication service - connect identity to one or more keys (long term identifier, key that can be used to auth protocol messages)
- delivery service - delivers messages in an async fashion and they are stored until the recipient becomes available; the clients publish a set of initial keys/ keying material, which is used only once; the users can be auth using the auth service

Group operations - 4 types of operations can be performed: welcome, add, remove, update (these are functions)

- group addition
- users publish the initialization keys to the delivery server and other clients can request these keys to create a group
- the requestee gets the initialization keys of the chosen participants and create a new group state
- then they will send the a welcome and then an add message to all of these participants consecutively, updating the group state after each addition
- if the a new participant is added, the process is repeated and the add message is broadcast to the rest of the group and the new member will perform an update after they were invited (recommended)
- updating
- changes the participant's leaf secret and the direct path from the leaf => backward secrecy on the participant's leaf secret
- updates depend on the application
- deleting
- a member sends the removal request with the index in the tree and the direct path from the leaf to the root is blanked and the tree is truncated at the rightmost blank leaf

Initial implementations - MLS++ Cisco - <https://github.com/cisco/mlspp>
- Molasses, Trail of bits - <https://github.com/trailofbits/molasses> - Melissa,
Wire - <https://github.com/wireapp/melissa>

4. Technologies used

- React
- node.js
- Redux toolkit
- Virgil security
 - browser code
 - docs
- socket.io
- Firebase
- Docker

React - <https://reactjs.org/> - React is “a JavaScript library for building user interfaces”. - This library was used to create the frontend of the application. There are interactions with the database on user authentication and attachment upload.

Redux toolkit - <https://redux-toolkit.js.org/> - For state management in the front end.

Node.js with express - <https://nodejs.org/en/> - <https://expressjs.com/> - These are used for the backend and relationship with the database.

socket.io - <https://socket.io/> - Makes the real-time communication possible.

Firebase - <https://firebase.google.com/> - Holds the data of the application, such as users with their conversations, messages and attachments.

Virgil security - <https://developer.virgilsecurity.com/docs/e3kit/fundamentals/supported-algorithms/> - This framework is used for end-to-end encryption for both the messages and the attachments. It also provides a key server where the user data (public keys) are stored.

5. The application

The application is a full stack web application, created with React, Node.js and the database and authentication logic are hosted by Firebase. For the real-time property of the messaging app, the front and the back are communicating using Socket.io, which is a package created over websockets. The key management and encryption/ decryption are managed using the VirgilSecurity framework.

The functionalities provided by the application are as follows: - Login/ Signup - the user can login to an existing account or create a new account with the email and a password, upon opening the application, and is redirected to the main application window. - Logout - the user is send back to the login page. - Display chats - the conversations of the logged user are displayed automatically, if any. - Select chat and send messages and attachments - once a chat is selected, the user can send messages to the recipient(s) and also see the previous messages sent and received - Add chat - a private or a group chat can be added. Here, the user can select whether they want it encrypted or not - Delete chat - each chat will have a delete button, which erases it from the database and it won't appear in the user's list anymore, but will still be visible for the recipient. In case of a group, a message will be displayed that the user left the chat - Delete message - each message will have a delete button and will be removed from the user's message list. It will be still visible for the recipient.

General flow

Use case diagram

The user is greeted with a login page. From this, they can create a new account or login, using an email and a password.

On the left side, the list of conversations for that account (it is empty if a new account was created) will appear, as well as buttons to add a new private or group chat. Clicking one of the buttons will open a form over the chat list and, for the private chat, will ask for the email of the recipient and to

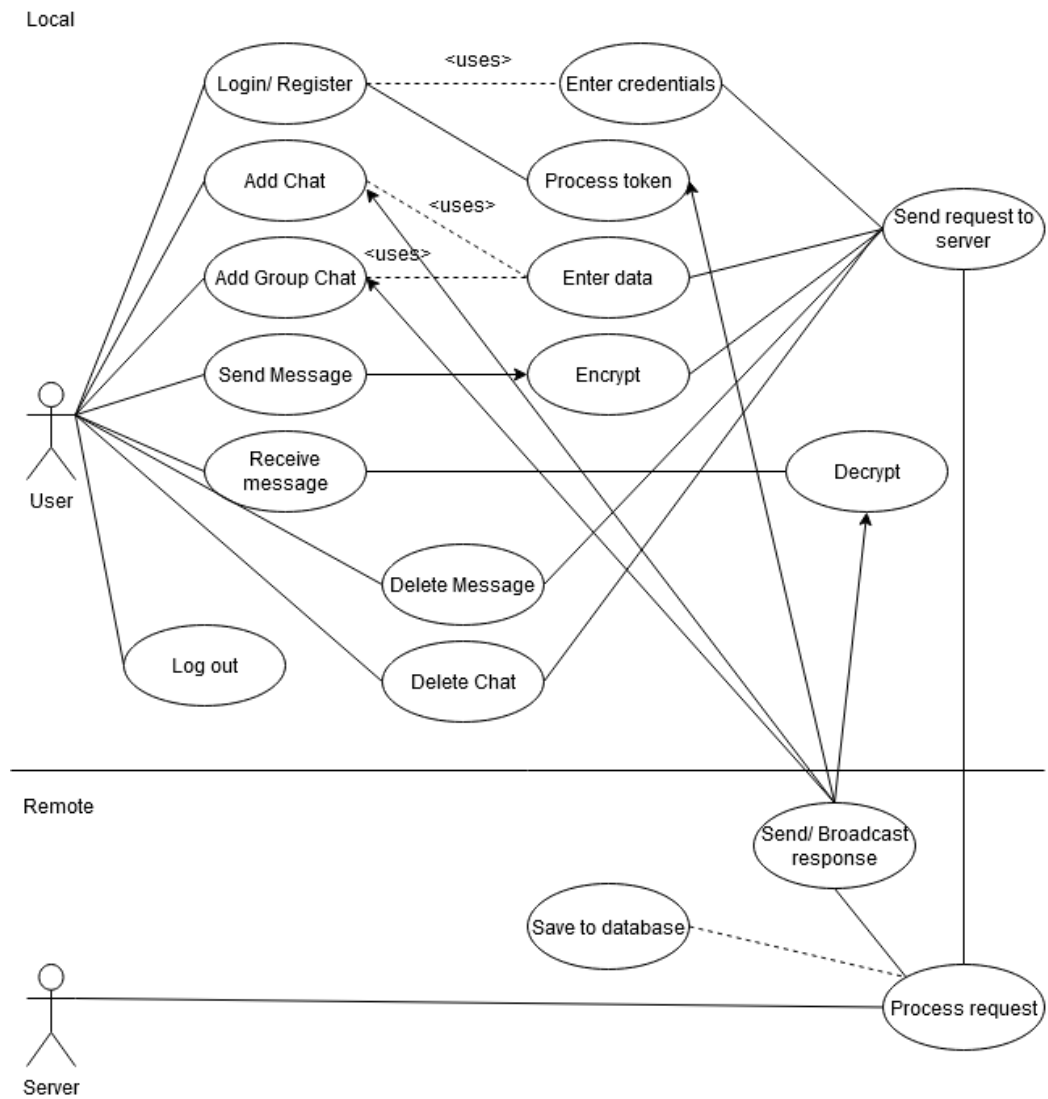


Figure 1: Use case diagram

choose if you want the chat unencrypted (encryption is selected by default) and for the group, there will be an additional input for the chat name and an button saying “Add another participant”. When this succeeds, the initiator and the recipients will be prompted with the newly created chat. The chats can be deleted by clicking the “x” button next to them and after confirming the decision in the dialog box. In case of groups, the participants will be notified if one of them leaves and will be deleted from the list of participants.

After choosing one of the listed chats (or after adding one), the user is prompted with a chat box and can send messages to the selected recipient. If the user wants to add an attachment, the file explorer will open and the client will be able to choose the desired file. Each message can be deleted by clicking “x” and confirming in the dialog box. These messages are only deleted for the current user, so they will remain visible for the rest of the participants.

The right side contains information about the user (the email), the participants in the selected group and the logout button. The logout button will redirect the user to the login page and all the locally saved data will be deleted.

Implementation and code layout

- diagrams for the frontend and backend are present in the “back.pdf” and “front.pdf” files

The frontend is created using React with functional components, the axios package to communicate with the server and Redux Toolkit for state management. The backend uses Node.js and express, to handle the requests. To obtain the real-time communication between the users, the socket.io library is used. When a message is sent or a chat is created, the server notifies the parties involved using sockets and only after the data is saved to the database.

The cryptographic infrastructure is mostly managed by the Virgil Security platform. When the user first registers, a profile is created in the database and, based on the email and password, a new “user card” is saved on the platform. This will be then requested on login and will provide the private key of the user, based on email and password, the public keys of the rest of

the users, on demand, and the encryption and decryption operations. Before each registration and login, the server needs to generate a token that will be used for authentication. These modified JSON web tokens are based on the user identity (email used for account creation) and they are composed of header, payload and signature. The header contains information about how the signature needs to be computed and the payload is the user data. The signature provides authenticity of the token and is made up of the header and the payload from before. They are encoded in base 64, joined with a dot and the HS256 signature algorithm is applied. /x

After the authentication token is obtained, the user token ? is requested by using the “initialize” method from the EThree class. It is then saved to the local context and the client can now use the application. The username and other user details are saved in the redux store.

A request is sent to the server to retrieve the chats from the database. When one of them is selected, the message list is populated and the user can send and delete messages. This data is also saved in the redux store, as well as the selected chat. The list of participants is shown on the right.

When a message is sent, it is first encrypted using the participants’ public keys and is signed with the private key of the sender, verifying that they are who they claim to be. Decryption happens in a similar fashion, by using the public keys and the message to be decrypted. Both of these operations make use of the **authEncrypt** and **authDecrypt** functions provided by the framework. The message is sent to the server and is saved to the database. The user gets a response with the id, so it can be handled easier on delete. ? The fact that the message was added is then broadcast to the user or group.

The decryption process takes place only when the messages are displayed to the user, so they are still encrypted in the store.

To add a new chat or group, the user needs to enter the email(s) in the corresponding form. The names are sent to the server and they are added to the database, if the emails are found, and the specified participants will have a new chat added and can communicate with each other. They are notified using sockets and the server is aware if this is a private or a group chat. When one of the users deletes the chat on their side, only the group is

notified using sockets and that participant is removed. Message encryption for groups uses the same functions and they are threatened in a similar way to the one-to-one messages.

On logout, all the locally saved data and the store are cleared and the user is sent to the login page.

6. Conclusions

7. References

-