DESIGNING SECURE MOBILE MESSAGING OVER THE INTERNET

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
Burak Kocuroğlu
January, 2016

DESIGNING SECURE MOBILE MESSAGING OVER THE INTERNET

By Burak Kocuroğlu

January, 2016

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

İbrahim Körpeoğlu (Advisor)

Ali Aydın Selçuk

Hakan Ferhatosmanoğlu

Approved for the Graduate School of Engineering and Science:

Levent Onural
Director of the Graduate School

ABSTRACT

DESIGNING SECURE MOBILE MESSAGING OVER THE INTERNET

Burak Kocuroğlu
M.S. in Computer Engineering
Advisor: İbrahim Körpeoğlu
January, 2016

Mobile messaging over the Internet is one of the most actively used communication methods. As it is heavily used for almost all kind of topics, the security of it becomes a major concern. However, there is no widely accepted security protocol standard for it. Each implementation either defines its own security protocol or adopts an existing one. We have defined a set of security requirements for secure messaging applications. Some of the most popular secure messaging applications (Cryptocat, Telegram, Threema and Signal) are analyzed according to these requirements. We have also designed our solution to match the requirements and improved its security as much as possible without harming the usability. Our solution provides E2E encrypted messaging with PFS support, local disk encryption, certificate pinning, improved random number generation with user input and uses a strong KDF. Our design rationales for the requirements are presented and discussed in detail.

Keywords: Secure mobile messaging, End-to-end encryption, Mobile messaging, Secure instant messaging.

ÖZET

İNTERNET ÜZERİNDEN GÜVENLİ MOBİL MESAJLAŞMA TASARIMI

Burak Kocuroğlu
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: İbrahim Körpeoğlu
Ocak 2016

İnternet üzerinden mobil mesajlaşma en sık kullanılan mesajlaşma yöntemlerinden biridir. Birçok farklı konuda sıklıkla kullanıldığı için, güvenliği de önemli bir hale gelmiştir. Ancak mobil mesajlaşma için kabul görmüş bir güvenlik standardı yoktur. Her uygulama güvenliğini kendi belirledikleri veya başka kaynaklardan uyarladıkları protokollerle sağlamaktadırlar. Biz de internet üzerinden güvenli mobil mesajlaşma için birtakım güvenlik gereksinimleri belirledik. En çok kullanılan uygulamalardan bazılarını (Cryptocat, Telegram, Threema ve Signal) bu gereksinimlere göre analiz ettik. Ayrıca bu gereksinimleri karşılayan kendi çözümümüzü tasarladık. Kullanılabilirliğine zarar vermemek koşuluyla güvenliğini mümkün olduğunca üst seviyede tuttuk. Çözümümüz, mükemmel iletme gizliliği (PFS) desteğiyle uçtan uca şifreli mesajlaşma, sertifika iğneleme ve kullanıcı girdisiyle geliştirilmiş rassal sayı üretimini desteklemektedir ve güçlü bir anahtar üretme fonksiyonu kullanmaktadır. Güvenlik gereksinimlerini sağlayan tasarımsal kararlarımız sunulmuş ve detaylı olarak açıklanmıştır.

Anahtar sözcükler: Güvenli mobil mesajlaşma, Uçtan uca şifreleme, Mobil mesajlaşma, Güvenli anlık mesajlaşma.

Acknowledgement

I would like to express my sincere gratitude to my advisor Assoc. Prof. Dr. İbrahim Körpeoğlu and co-advisor Prof. Dr. Ali Aydın Selçuk for their supervision, support and patience during the study of this thesis. I would like to thank my thesis committee member Prof. Dr. Hakan Ferhatosmanoğlu for his time and valuable inputs.

I also would like to thank Özlem Başak Baltaşı for reviewing the drafts and my family for being very supportive and patient. Finally, I am grateful to Bor Software for their support.

Contents

List of Figures

Chapter 1

Introduction

Mobile text messaging has been around for more than 20 years as Short Messaging Service (SMS). It is a widely accepted form of communication. However, as mobile Internet usage increases and becomes available to more and more users, sending messages over the Internet became a more attractive alternative. Billions of people started to use messaging applications like WhatsApp, Viber, Line, WeChat or similar ones. These mobile messaging applications are usually free and offer text messaging, photo, video and audio sending, geographical location sharing, etc. This cheaper and feature-rich alternatives started to take the leadership from SMS. In 2012, more instant messages are sent than SMS [7] and the gap is increasing ever since.

As we rely on mobile messaging, we have started to use it extensively and not only for text messages but also for media and other data as well. For example, many people find it easier to send a photo using WhatsApp than to send it via e-mail. Like that, text messages, photos, videos, current location information are being exchanged via mobile messaging applications. These messages and media may contain lots of sensitive and private information. The security of these applications becomes critical as their roles in our lives get bigger and new threats are discovered. Especially after Snowden leak [8, 9, 10], we realized that we have been spied on with mass surveillance programs. And now, a lot of people started

to pay attention to the security of their communication channels.

When Secure Sockets Layer (SSL) was not as common as today, many messaging applications were using plaintext HTTP connections to send and receive messages. If these unencrypted HTTP connections are used, anybody who has access to the network can read the messages. Also, with such systems, servers already have all of the message contents without doing any attacks. With SSL, the connections between clients and servers are encrypted so third parties cannot access the communication data easily. Today, many messaging applications' security relies on simply using client-to-server encryption. Although this is secure against network listeners and other possible third parties, servers still have access to the data. They can access the private messages, analyze it for marketing purposes, serve it to third parties and pretty much can do anything with it without users' knowing. We have seen PRISM [11, 12, 13] which is a behind-the-doors mass surveillance program of United States National Security Agency (NSA). It had been operating since 2007. It has a scary list of provider companies which "allegedly" provide their users' data to the program. The list includes Microsoft, Yahoo, Google, Facebook, PalTalk, AOL, Skype, YouTube and Apple. Of course we cannot be sure if these allegations are true but they surely seem feasible.

In addition to evil servers and governments, certificate authorities (CA) may also pose danger. They are "trusted" but they have the ability to generate fake *valid* certificates for other services. A CA's intentionally doing this may be regarded as a far-fetched conspiracy, but there have been cases [14, 15, 16] where CAs are compromised or "misused". For common use scenarios, CAs' trust establishment method serves an important and fundamental role and using them may be acceptable. However, for security-critical systems they shouldn't be relied on. Because their security depends on the weakest link, compromise of only one root or intermediate CA puts the whole system in danger.

If servers, CAs and the communication channel are not trusted, an end-to-end encryption (E2E) protocol is needed for securely exchanging messages. With E2E encryption, data is encrypted at one side and only decrypted at the destination side. For example, If Alice is sending a message to Bob, Alice encrypts the

message, sends it to the server and server delivers the encrypted message to Bob. Only Bob decrypts the message and can see its contents. Server or other third parties cannot decrypt the message. Using E2E encryption does not mean the communication is secure. It also has to be designed and implemented securely.

In this thesis, we have analyzed state of the art in secure messaging and proposed our design as a complete secure messaging solution. There are many messaging protocols which offer E2E encrypted secure messaging. They have different security and usability features. We have analyzed most actively used and well documented secure messaging applications and their protocols. Their security features, possible weaknesses and design rationales are presented.

Our aim was designing a secure messaging protocol without harming usability. Generally, lack of usability drives away the user from secure systems. We wanted to give the best security without changing usability and provide additional options for advanced users. We presented what security features are required against known attack surfaces and how they should be implemented to provide a secure messaging experience. We have tried to combine the best of existing security protocols and improve them where possible. We also designed the solution with an enterprise-friendly view so that it can be used not only by individuals but also enterprises.

The remaining parts of this thesis are organized as follows: Chapter 2 gives detailed background information about cryptographic fundamentals of secure messaging. Chapter 3 analyzes existing secure messaging protocols. Chapter 4 presents our design and Chapter 5 concludes.

Chapter 2

Background

In this chapter, we described common security topics which are used in secure messaging. These topics' relations to secure messaging are briefly explained.

2.1 Symmetric Encryption

Symmetric encryption is the encryption method where the same key is used for encrypting and decrypting the data. Thus two parties must have the same encryption key (shared secret) to successfully transmit messages. Some well known examples include AES, Blowfish, RC4 and 3DES.

In messaging, symmetric encryption usually resides at the core of whole communication system. The most common scenario is Alice and Bob share a secret key, Alice encrypt the message with the key and sends it to Bob, then Bob decrypts it with the shared secret key. Here, Alice's encryption and Bob's decryption uses symmetric encryption (same key is used for both encryption and decryption).

2.2 Asymmetric Encryption

At asymmetric encryption or Public-key cryptography, two different but related keys are used for encryption and decryption. The keys are called public-private key pair; public key is distributed and private key is kept secret.

There are two common usages of asymmetric encryption:

1. Encryption/Decryption

   If the message is encrypted with recipient's public key, it can only be decrypted with recipient's private key. In this scenario, if recipient publish his public key, anyone can send him encrypted messages which can only be decrypted by him.

2. Signing/Verifying

   If the message is signed with a private key, it can be verified with its corresponding public key. Thus when trying to verify using a public key, if it succeeds we can be sure that the data is signed with someone who has the related private key.

In messaging, asymmetric encryption can be used for both encryption/decryption and signing but instead of using directly on the message, asymmetric encryption is used on the process to generate secret keys. Especially OTR-like protocols use it to sign the parts of secret key, so both parties can be sure they are talking to correct person but messages still provides repudiation.

2.3 Message Authentication Code

Message authentication code (MAC) provides integrity and authenticity checks for messages. MAC functions usually take a secret key and the message content then output a rather short tag of the message. Receiving end computes the tag

with same secret key and the message; if the tags are same then he can be sure of that the message is not tampered with. If the message content is changed, the output would be different (integrity check) and if the sender does not have the secret key then he wouldn't be able to produce the tag (authenticity). The most popular MAC algorithm is keyed-hash message authentication code (HMAC) [17].

Verifying the integrity of messages is required for messaging systems against message content tampering, so MACs play an important role in messaging systems.

2.4 Diffie-Hellman Key Exchange

Diffie-Hellman Key Exchange (DHKE) [18] is a method to decide on a shared secret key over an insecure channel. DHKE requires no previous information about parties to establish a shared secret. The shared secret can be used for symmetric encryption.

In DHKE both parties need to have key pairs. One side's private key is combined with other side's public key and a shared secret key is generated.

- a: Alice's private key
- b: Bob's private key
- A: Alice's public key = $g^a \bmod p$
- B: Bob's public key = $g^b \bmod p$

1. Alice sends Bob her public key: A = $g^a \bmod p$
2. Bob sends Alice his public key: B = $g^b \bmod p$
3. Alice calculates $B^a = (g^b)^a \bmod p = g^{ab} \bmod p$

4. Bob calculates $A^b = (g^a)^b \bmod p = g^{ab} \bmod p$

They both have $g^{ab} \bmod p$ secret and neither of them can derive other's private key. Also, even Eve has listened the conversation and has $g^a \bmod p$ and $g^b \bmod p$, she cannot produce $g^{ab} \bmod p$

Elliptic Curve Diffie-Hellman: Elliptic Curve Diffie-Hellman (ECDH) is a variant of Diffie-Hellman. It uses elliptic curve cryptography for shared secret generation. Its main advantage is that it provides similar key strength with smaller key sizes. For example, 160 bit key size provides similar strength to 1024 bit RSA and 224 bit key size provides similar key size to 2048 bit RSA[19].

In end-to-end encrypted messaging, usually DHKE or its derivatives are used to create a key which is then used to encrypt messages between parties.

2.5 Perfect Forward Secrecy

Perfect forward secrecy (PFS) is a property which ensures compromising a private key now would not reveal past communications. Or as its name suggests, a future private key's compromisation would not reveal current communication's content. To provide this capability, same key shouldn't be used over and over. In messaging context, PFS provides the trust of losing your keys sometime (in the future) would not reveal all of your past communications. For example, if a server which is vulnerable to Heartbleed [20] bug is compromised, its private key may be stolen. In this scenario, if the server's TLS/SSL implementation supports PFS, previous communication data would be safe from attackers.

Perfect forward secrecy is a critical feature of messaging systems, especially when messaging history spreads to a long time. If PFS does not exist, then a compromise at anytime during the messaging lifetime may cause the leak of all messaging history. When using a secure messaging system, we want the messages to be secure even after they are transmitted.

2.6 Key Derivation Function

Key derivation functions (KDF) are used to derive keys from passwords or other keys. Usually the input is stretched until desired length is obtained. This stretching method does not introduce any more security to the derived key. And same input value always derives the same key if same KDF is used.

KDFs are especially useful for converting user supplied passwords to encryption keys, since user supplied passwords cannot be used directly as encryption keys if they are not exactly in the same format with expected encryption key format.

Derived keys are only as secure as supplied passwords. And passwords aren't as strong as full-length randomly generated keys. Because of that, encryption systems which use derived keys can be subject to brute-force attacks. To harden the system against brute-force attacks, KDF function can be made more time and memory consuming. PBKDF2 [21] is probably the most common KDF function. It includes an iterations parameter which is used to set computation power. bcrypt [22] also has a cost parameter. It is used to determine the key expansion rounds. If these cost parameters are increased, computation power required to derive the key is increased. This makes it harder to brute-force attacks by making them more time consuming. The cost parameter should be selected such a way that it wouldn't be a burden for normal users and still strong against attacks. Although both bcrypt and PBKDF2 have cost parameters, bcrypt has the advantage of being more resistant to brute-force attacks with graphics processing units (GPU).

Increased CPU cost helps against brute-force attacks, however, parallel computing and application-specific integrated circuit (ASIC) or field-programmable gate array (FPGA) makes brute-forcing relatively easier. A further improved method against these brute-force specific hardware devices is scrypt [23]. scrypt also utilizes the memory usage with a parameter. If memory usage of the operation is high, then it is much more harder to produce or use specific cracking hardware for it.

In messaging, KDFs are usually used to generate the key for securely storing local messages. Using weak KDF may result in compromising of the messages in case of lost device.

2.7 Certificate

A certificate is a signed public key which also includes information about the owner and the signer. These certificates can be used for trust establishment over an insecure channel and are also vital components of TLS. A typical X.509 [24] certificate contains serial number, owner and signer info, algorithms used, validity period, public key and thumbprint of itself. If the validity period has not expired and the signer is trusted and then the certificate is also trusted.

As per X.509 public key infrastructure (PKI) model, certificates are obtained from certificate authorities (CA). After a CA verifies the claim of certificate requester, it signs the certificate. The claim can be about the ownership of a domain and/or representing a company etc. The cost of certificate signing varies according to the claim type and certificate authorities.

Certificates' most common usage is for verifying the application servers and carrying out a TLS handshake for further encrypted communication between client and server. If certificate pinning is used, there is no need for a CA to sign the server certificate because it is only assumed valid if it matches the pinned certificate in the application code. Like servers, clients can also use certificates (called "client certificates") for identifying themselves. For example, during a TLS handshake, server can request client certificate from the client and verify the identity of the client. Usage of client certificates aren't common since it is unlikely for individuals to have certificates.

2.8 Web Public Key Infrastructure

Public key infrastructures (PKI) set rules for the management of public key authentications. According to PKI rules, certificates are created, distributed, authenticated, used and revoked. There are different types of PKIs like X.509 PKI [25] (Web PKI), Web of Trust, simple public key infrastructure (SPKI) etc. X.509 PKI is the most actively used one and it is used by browsers and applications as default PKI.

Web PKI defines a set of trusted root Certificate Authorities (CA). These root CAs sign and thus trust a larger set of intermediate CAs. Either root CAs or intermediate CAs sign the certificates of entities. When someone question whether a certificate is valid, he/she checks if the certificate is signed by a trusted CA. If the signing CA is an intermediate CA, then its signer CA is checked. Using this chain of trust, a certificate's validity is decided. The list of trusted CAs are pre-installed on users' browser/device (they could be in the browser or directly at operating system). To support flexibility, in most systems, there are options to disable existing CAs or add new ones. Certificates also have a validity period, after which they are considered invalid. Besides expiring, some certificates may be compromised, in that case they are revoked using certificate revocation lists (CRL).

Today, having your certificate signed usually costs money. The price changes according to the popularity of CA, trust coverage of different browsers and validation levels like "Extended Validation" which validates the company or organization as well as the domain. There is also a free and automated CA called "Let's Encrypt" [26] within "Linux Foundation Collaborative Projects". "Let's Encrypt" offers free signing of SSL certificates. It does not require any human interaction for creating certificates, only ownership of the domain is required and a challenge is requested for proving private key validation. As of December 2015, it is still in public beta but it is trusted by all major browsers [27].

From users' perspective this system may work seamless; certificates are checked

against predefined authorities and when a certificate is not trusted by the certificate authorities, user is alerted. The system works very well if all of the CAs can be trusted all the time. However, there are concerns about the CAs' security because a compromised CA puts the whole system in danger. And this has been happened before [14, 15, 16]. There are some solutions for this like Certificate Pinning (See Section 2.10) or alternatives to X.509 PKI's CA model. The alternatives are out of the scope of this thesis, however, we inspect the usage of certificate pinning.

2.9 Transport Layer Security / Secure Sockets Layer

Transport Layer Security (TLS) is the successor of Secure Sockets Layer (SSL) but sometimes "SSL" is used as when talking about the newer standard TLS. SSL's history goes back to 1995 and TLS' first version is introduced in 1999. Now, using TLS/SSL is the de facto way of securing communications over the Internet. Without TLS/SSL, connections to web sites or e-mail servers can be interfered easily. This can be done by Internet service providers or even by someone who listens network traffic if local network security is not strong enough. Firesheep [28], a Firefox extension was released in 2010 to demonstrate how easy it is to get cookie data if TLS/SSL is not used. When activated, it listens the network and collects popular web sites' cookies. Using the cookies, one can log in to its web site. The extension has showed the importance of using TLS/SSL and probably helped many web sites to switch to TLS/SSL.

With TLS, both sides can be authenticated by using public key cryptography. Usually only server is authenticated, but clients can also be authenticated using client certificates. Symmetric encryption is used for securing communication data. Encryption keys are created after a handshake between the sides. TLS can also support perfect forward secrecy if it is configured for it.

The general approach for establishing trust is based on Web PKI (See Section 2.8) but it is not mandatory to use it. Especially it is common for security-oriented systems to use certificate pinning (See Section 2.10) instead of Web PKI's certificate authority model.

Many applications and websites use TLS whether they are security oriented or not. Almost all platforms and HTTP libraries support it. Since the encryptions are handled at transport layer, usually no application level change is required.

2.10 Certificate Pinning

TLS' current trust model requires trusting some set of trust notaries (Certificate Authority, CA). If any of the trusted CAs are compromised or inherently evil, the whole system becomes vulnerable. We have seen many examples of this [14, 15, 16]. Certificate pinning tries to solve this problem on application basis. Server certificate's public key is defined in application code and only that certificate is trusted by the application. Thus CAs are not used, only the server's certificate is used and CA compromisation or evil CAs don't affect the communication.

Another pinning method is trusting only a specific CA. Thus compromisation of other CAs does not affect the security of the communication and handling certificate updates are easier as long as same CA is used.

A general downside would be updating the certificate. If server changes/updates its certificate, new certificate's descriptive information must be updated at all clients as well. Certificate updates can be because of expiration, compromising or other possible causes. If the same certificate signing request (CSR) is used (this requires using the same private key), certificate pinning data at clients don't have to be updated. However, if the private key is compromised then pinned certificate data should be updated as well. This may be costly for mobile applications since users may not be opted in for automatic updates. Pinning a CA solves this issue as long as same CA is used always. Of course pinning a CA does not fully solve

CA compromisation problem only narrows it to one CA.

Certificate pinning is a measure against malicious CAs or mistakenly signed certificates. Both cases may have valid scenarios in secure messaging environment which may leak sensitive data to governments, companies or Internet service providers (ISP).

2.11 End-to-End Encryption

With end-to-end encryption, data is encrypted at one end point and only decrypted at the receiving end point. Thus no other party (like Internet service providers or application servers) can view or modify the data. End-to-end encryption is essential for private or security-critical communications if the transfer channel is not trusted.

Common websites or services advertise that they use encryption but it is usually between client and server which protects the data against Internet service providers or network sniffers but application server can still have access to the data. For example, when sending an e-mail using any commodity e-mail providers (Gmail, Yahoo, etc.) your e-mail content cannot be read by any third party but can be read and modified by the e-mail providers.

If two parties share a secret key, then simply encrypting at one side and decrypting at other side would provide end-to-end encryption. If there is no shared secret key between parties, one side can encrypt using other side's public key, and the other side would decrypt using his private key. The same principle applies for the other direction as well. However, using public-key encryption each time is costly, so usually a shared key is generated with Diffie-Hellman key exchange (see Section 2.4) using public and private keys of both sides and then the shared key is used for symmetric encryption.

End-to-end encryption is the core concept of secure messaging. Without it,

application servers can read or modify the message content.

2.12 Off-the-Record Messaging

Off-the-Record Messaging (OTR) [29] is a messaging protocol designed by Ian Goldberg and Nikita Borisov in 2004. Since its introduction, many libraries and application have used OTR. It provides end-to-end encryption, perfect forward secrecy (PFS) and repudiable authentication.

Message encryption keys are created with Diffie-Hellman key exchange (DHKE) but instead of directly using public/private key pairs random numbers are used as DHKE parameters. The numbers are signed by private keys of each side so DHKE parameters are authenticated not the messages (See Figure 2.1). This provides repudiable authentication because messages or their encryption keys are not signed by sender. Both receiver and sender could have been created a specific message. To increase possible signers, old message encryption keys are published.

Alice and Bob pick random x, y resp.
A $\rightarrow$ B: $g^x, Sign_{Alice}(g^x)$
B $\rightarrow$ A: $g^y, Sign_{Bob}(g^y)$
SS=g^{xy} a shared secret

Figure 2.1: OTR Diffie-Hellman Key Exchange [1].

Today many applications use OTR or similar protocols inspired by it. Some of the popular applications which use OTR are: Cryptocat [30], ChatSecure [31] and Jitsi [32]. Axolotl [33] protocol - which is used by "Signal Private Messenger" - is also inspired by OTR.

Chapter 3

Analysis of End-to-End Secure Messaging

This chapter surveys previous work in secure mobile messaging. Some of the most widely used secure messaging applications are selected and their protocols are analyzed thoroughly. We have searched for applications which claim to be a secure messaging application. We found Signal Private Messenger (previously TextSecure), Threema, Telegram and Cryptocat to be widely used and well documented. These applications' security is analyzed throughly. When analyzing an application, at first, an overview is presented: its short description and what security features it has are explained. Then, we have analyzed it in the light of following topics:

Registration: Before using messaging services, clients usually need to register to application server by providing some descriptive information about themselves. Sometimes this registration phase may require some kind of authentication like phone number verification. Most of the applications which target mobile devices use phone number verification in order to register. Using phone numbers as unique identifiers has the advantage of importing or using someone's phone contacts as message contacts. Otherwise another contact list for messaging needs to be managed.

After the registration is completed, devices will have some kind of identifiers to authenticate themselves and the server can recognize devices for future requests. Within this topic, we will analyze how a new client is included in the system, how it is authenticated for the first time and what outcome will be generated for both the server and the client.

Authenticating the Server: Clients connect to application servers for message sending and receiving. However, clients need to be sure if they are connecting to right server and no man-in-the-middle (MitM) attack takes place. For this purpose, usually TLS/SSL is used. As the trust model of TLS, clients connect to the server, using server's host name (or IP), retrieve server certificate, check the certificate against host name, check validity period and check if the signer of the certificate is trusted. If these conditions are matched, clients trust the server.

With TLS' trust method, there is a problem: when checking a certificate, clients look if certificate is signed by *any* trusted certificate authority (CA). This means if a trusted CA is compromised, it can sign any certificate, and they will seem valid by clients. We have seen many examples of this [14, 15, 16]. To overcome this issue, certificate pinning is used: server's certificate is included in the application and when connecting to the server, server's certificate is checked against this pinned certificate. Thus trusting to all *trusted* CAs is avoided.

When analyzing a protocol, we have examined how clients authenticate the server. Is TLS used? Is certificate pinning used? Is MitM possible during messaging and at the registration phase?

Authenticating the Client: In order to send messages, clients need to be authenticated by the server. This could be done via a few methods: using username-password, an identification token, public key cryptography etc. These different methods have different advantages and disadvantages. We have analyzed various protocols' client authentication and presented their security levels.

Client-to-Client Authentication: When two parties are messaging with each other, they need to know if they are speaking to correct person. Since

clients don't interact with each other directly, messages are transmitted over the server. And with this setup, clients don't authenticate other users; server is responsible for forwarding the messages to correct users. However, if the server is compromised or inherently evil, man-in-the-middle (MitM) attacks can be easily carried out. To detect if a MitM attack exists, users should be able to check fingerprints or compare session keys manually. The most common use case of this is: when Alice and Bob first exchange messages, they compare their fingerprints through a secure channel so they know they are talking to each for sure. This secure channel could be over the telephone line or in person if that's possible. After this first handshake, the contact is paired with the fingerprint. Then if someone's fingerprint changes, user will be notified, so he/she will know something is not right: either someone is interfering or the other party is updated his/her signature. This fingerprint checking is an important feature to avoid MitM attacks for applications which support end-to-end encryption.

In end-to-end encryption, verifying the other party is needed to be sure that you are not messaging with an imposter. Thus, we have included a client-to-client authentication analysis topic and checked if the protocol allows checking other party's identity.

Message Encryption & Perfect Forward Secrecy: Messages are encrypted to block unauthorized third parties from accessing to its contents. Message authentication is used to verify that the message is sent by a trusted sender and to check if the message is altered. Both are critical for securely delivering a message. Usually other parts of the protocols' design shape to provide the keys to encryption and message authentication process. In this topic, we have listed encryption and MAC algorithms of the protocols and looked at their parameters to see if they meet general security standards. Perfect forward secrecy support of the message encryption scheme is also analyzed and its details are explained.

Local Storage: Most of the time messaging applications save messages to a persistent storage. So when users are messaging, they can see where they left previous conversation or search through old messages. According to the protocol design, unread messages can also be saved on disk. If those messages are not

encrypted, they can be retrieved in case of losing the device's possession. It is important to encrypt the messages properly. A proper local storage encryption should not store plaintext key materials on disk and use strong ciphers with enough key lengths.

Some applications save messages for a limited time then the message is automatically deleted (self destruct). From the end user perspective, this may look like a secure way to temporarily store messages, however this method does not provide real security without further actions. Messages are still vulnerable until they are deleted and we cannot be sure the client of other side really deletes the messages. On hardware level, when data is written to disk, just removing it using standard methods does not actually deletes it from the disk, just marks it as deleted. A recent research [34] carried out by Avast [35] shows how much data can be retrieved from used smartphones. "Self-destruct" or similar features are only mentioned as a usability feature since they don't offer much security value.

There are secure delete tools for classical hard disk drives (HDD) but for solid-state drive (SSD) these won't work reliably [36] because of some of their properties like *wear leveling*. Today most smartphones use SSD or its variants but none of them uses HDD (as far as we know). So we can say that secure deletion is not a solution for smart phone users. Whether temporary or not, only secure way to save critical data on disk is using encryption. Because of that, we have analyzed how messages are stored on disk within this topic. Are they encrypted? Where is the encryption key stored? Are security blocks configured correctly?

Message History Transport: When switching to another device, it is usually desired that messaging history is not lost. Also, users may want to take backups of their messages in case of hardware failures or lost devices. Within this title, we analyze how the applications handle this scenario and the security of this process. Popular approaches are synchronizing the messages using central server, manually taking backups and restoring them on new devices or not provide any option at all.

Public Key Change: It is important to alert the users when the other party's

public key changes. This may mean a man-in-the-middle attack taking place and the flow shouldn't continue as before. Public key change can also happen if a user reinstalls the application. We tested the applications to see how they behave when a contact's public key changes. We expect there should be enough differences so the user can know if something unexpected is going on.

3.1 Cryptocat

Cryptocat [30] is an open source end-to-end encrypted messaging solution. It is implemented as browser extensions (Google Chrome, Mozilla Firefox, Apple Safari, Opera) and iPhone application. Its initial release goes back to 19 May 2011.

Cryptocat uses Off-the-Record (OTR) Messaging protocol [29] for two-user chats. For group chats, its own protocol Multiparty Protocol Specification [37] is used. OTR's end-to-end encryption and deniability is provided for one-on-one user chats. However, group chats don't support deniability. Perfect forward secrecy is provided by generating new key pair at every application start.

3.1.1 Registration

There isn't a traditional registration phase to use Cryptocat. Users only need to type a conversation name and nickname to enter a chat room. While entering or creating the chat room, a new key pair (256 bit Curve25519) is generated. Besides the main server, there is also a chat server which uses Extensible Messaging and Presence Protocol (XMPP). To register to the chat server, 256 bytes long random alphanumeric user name and password are created. After registering to chat server, users' public keys are sent to the server and associated with their usernames. Same procedure is applied whenever a new session is started (i.e. browser window is closed and then opened again). To be safe from man-in-the-middle attacks, users need to verify others' public keys and they need to do it

every new session.

3.1.2 Authenticating the Server

Cryptocat uses TLS to connect to the server and server's certificate is signed by an trusted certificate authority (CA) but it is not pinned in the application code. Chromium browser has its certificate pinned [38], though. For other browsers, CAs can do man-in-the-middle attacks and CA compromises may be a problem.

3.1.3 Authenticating the Client

Each time when the application is started, a new public/private key pair and username-password are generated. These username-password and public/private key pair are associated at the end of registration step. Cryptocat's main server uses public/private key pair to authenticate the users and XMPP server uses randomly generated 256 bytes long username and password to authenticate clients.

3.1.4 Client-to-Client Authentication

When users start a new chat, they can verify each other by comparing their fingerprints over a secure channel. Also, at fingerprint display screen there is a secret question / secret answer area where one can ask the other party a secret question and check if the other party has given the expected answer. The asking side only sees if the other side has given the expected answer or not.

Encryption keys and fingerprints change at every application restart, so this operation should be done at every chat. With each session, finding a secure channel to compare fingerprints is not a very feasible option and there is an open issue [39] about that on GitHub [40].

```
{
   "type":"message",
   "text":{
      nickB:{
         "message":ciphertextB,
         "iv":ivB,
         "hmac":hmacB
      },
      nickC:{
         "message":ciphertextC,
         "iv":ivC,
         "hmac":hmacC
      }
   },
   "tag":tag
}
```

Figure 3.1: Cryptocat Message Format.

3.1.5 Message Encryption & Perfect Forward Secrecy

At one-on-one chats, Off-the-Record (OTR) messaging protocol is used with an open source OTR JavaScript implementation [41]. The library supports Off-the-Record messaging protocol version 3 [42]. Cryptocat uses library's default 1024 bit keys.

For group chats Cryptocat's own protocol Multiparty Protocol Specification [37] is used. This protocol does not provide deniability and PFS is only provided because of changing key pairs at every chat. However, migrating to Multi-party Off-the-Record Messaging (mpOTR) is planned [43]. Currently, users generate pairwise shared keys. When a message is being sent, it is encrypted for all other members of the group. Then all of the encrypted messages and a tag are sent altogether. Message format is defined in Figure 3.1.

Shared keys are generated with Curve25519 512 bits; 256 bit for AES-CTR-256 encryption and 256 bit for HMAC-SHA512. During the encryption, random IVs are used. An IV array is kept by clients, so no IV will be re-used. Messages are

encrypted with first half of shared keys (256 bit) using AES-CTR-256. 16 byte IVs are used during the encryption. When sending a message, it is encrypted separately for all of the receivers and its ciphertext, IV and HMAC are included in the message for all recipients. HMAC is generated using the last 256 bits of sharedSecretNM. Following concatenation is processed with HMAC-SHA-512:

```
ciphertextAlice || IValice || ciphertextBob || IVbob ||
ciphertextCarol || IVcarol || ...
```

To ensure everyone is getting the same message, a tag is computed by concatenating plaintext and all recipients' HMAC. *tag* is computed by applying 8 times SHA-512 of the following concatenation:

```
plaintext || HMAC-SHA512alice || HMAC-SHA512bob || HMAC-SHA512carol ...
```

This message encryption scheme doesn't provide perfect forward secrecy (PFS). However, by re-generating the keys at every application start, session based PFS is provided.

3.1.6 Local Storage

Messaging history is not saved, so local disk encryption is not needed for messages. Public/private key pairs are generated at every application restart, and they do not need to be stored either, so there is no local storage encryption mechanism at all.

3.1.7 Public Key Change

Since public keys of contacts change at every application restart, users are expected to verify them at every chat session. And if the public key changes during the chat, it doesn't affect the conversation until the next chat session is started.

Because session keys are generated at the start of chat and then they are used for encryption. Re-keying would update them with changed private/public key pair but it isn't implemented.

Verifying contacts does not persist across chats, so whether the public key of other party is changed or not, users see it as not verified for new chats. So either they should note the fingerprint using an external tool and check it when a new chat starts or verify the other party through a trusted channel.

3.2 Telegram

Telegram is founded in 2013 as an independent and nonprofit company by Nikolai and Pavel Durov brothers who are the founders of Russia's largest social network, VK [44]. It has its own custom protocol which uses client-to-server encryption as default but also supports end-to-end encryption.

Telegram has iOS, Android, Windows Phone applications and also web client and desktop (Mac OS X and Windows/Mac/Linux) versions. The clients use Telegram API [45] and anyone is free to use it to develop their own clients. Telegram clients are open source but the server code is proprietary. In their FAQ page [46] they say, they also plan to open source server side ("*All code will be released eventually*" as an answer to "*Why not open source everything?*" question).

Telegram has collected a serious amount of users. As of December 2015, its Android application has a download count between 50.000.000 and 100.000.000 at Google Play [47].

Telegram uses a custom protocol (MTProto) on top of HTTP or TCP (HTTPS is not used). This custom protocol provides client-to-server encryption, certificate pinning and optional end-to-end encryption. Server's public key is pinned at the clients to avoid MitM attacks. If end-to-end encryption is not used, messages are

stored at Telegram servers.

Telegram has two different messaging modes: normal chats and secret chats. Normal chats only provides client-to-server encryption while secret chats provides end-to-end encryption. The default option uses normal chats. When using normal chat, message history (messages, photos, video files etc.) is stored at the server to provide backup and synchronization between devices. To use secret chat, it needs to be specifically initiated and two parties must be online at the same time for key exchange. Secret chats also provide Perfect Forward Secrecy (PFS) by re-keying every 100 message or every week, whichever is reached first.

3.2.1 Registration

During registration, an authorization key (*auth_key*) is generated at both client and the server by using Diffie-Hellman key exchange. These keys are device specific and at the end of generation, it is made sure that no two devices have the same key. This authorization key is used for almost all API calls, only a small portion of the API is available unregistered users.

Authentication Key (*auth_key*) Generation:

1. Client sends *req_pq* message which contains a generated nonce.
2. Server responds with *resPQ* which includes: *nonce*, *server_nonce*, *pq*, *server_public_key_fingerprints*. Here, *server_nonce* is randomly chosen by the server, *pq* is the product of two different odd prime numbers, *server_public_key_fingerprints* is a list of public key fingerprints. From now on all messages include (*nonce*, *server_nonce*) pair.
3. Client decomposes *pq* into prime factors.
4. Client sends *req_DH_params* message: *nonce*, *server_nonce*, *p*, *q*, *public_key_fingerprint*, *encrypted_data* where *encrypted_data* contains

new_nonce which is generated by client *encypted_data* can also contain an optional *expires_in* parameter which is used to provide PFS.

5. Server responds with: *nonce*, *server_nonce*, *encrypted_answer* *encrypted_answer* contains *new_nonce_hash*, g, g^a

6. Client sends *set_client_DH_params*: *nonce*, *server_nonce*, *encrypted_data* where *encrypted_data* contains g^b

7. At this step a DH key exchange is completed. *auth_key*=g^{ab}

Telegram uses phone number as a unique user identifier, and it needs to be verified at the registration. Users enter their phone number, then a 5 digit confirmation code is sent to that number via SMS. The application automatically reads incoming SMS and sends that code to the server to verify the phone number. After the phone number is verified, registration phase is completed. From the privacy and user experience perspective, one of the most irritating feature of this registration phase is that your registration information is displayed as "*user name* joined Telegram" on all of your contacts who use Telegram.

3.2.2 Authenticating the Server

Usually TLS/SSL is used to connect to the servers and servers' certificate is checked against trusted CAs. If more security is desired and trusting CAs is not desired then server's certificate is pinned at application. However, Telegram doesn't use TLS/SSL, instead uses its own custom protocol MTProto. Server's certificate is still pinned against man-in-te-middle (MitM) attacks.

With MTProto, *auth_key* is used for data exchange and authentication. This *auth_key* is generated at registration step (see Section 3.2.1) and used at every request by the client to check for the validity of the server. However, before creating the *auth_key* clients need to know if they are talking to the correct server. This is done by certificate pinning. At first connection to server, server's

certificate is checked against the pinned certificate to prevent MitM attacks. This way clients can know if they are talking with the correct server.

3.2.3 Authenticating the Client

As described in Registration section (see Section 3.2.1), an *auth_key* is generated using a Diffie-Hellman-like protocol. After creating *auth_key*, user's phone number is verified and associated with the *auth_key*. Telegram uses its custom protocol MTProto with this *auth_key* for all future requests to authenticate the client.

3.2.4 Client-to-Client Authentication

Users may want to be sure that they are talking to correct person and no man-in-the-middle (MitM) attack is taking place. Telegram allows this verification for secret chats. For normal chats, the server is completely trusted and even tough you verify the other side, server can view and modify the messages.

To start a secret chat session, two users must be online at the same time to generate a shared key. The key is generated using Diffie-Hellman key exchange protocol. The key is visualized using a 128-bit white-blue QR code like picture as in Figure 3.2. Two parties can compare these pictures and know if a MitM attack exists.

3.2.5 Message Encryption & Perfect Forward Secrecy

For encrypting messages, Telegram uses a custom protocol which is called MTProto. MTProto encryption mainly relies on *auth_key* which is generated via Diffie-Hellman (shared between client and server). *auth_key* is fed into a KDF then the key for message encryption is obtained. Message (plus salt and session id) is encrypted with this key using AES-IGE encryption. Message's hash is also

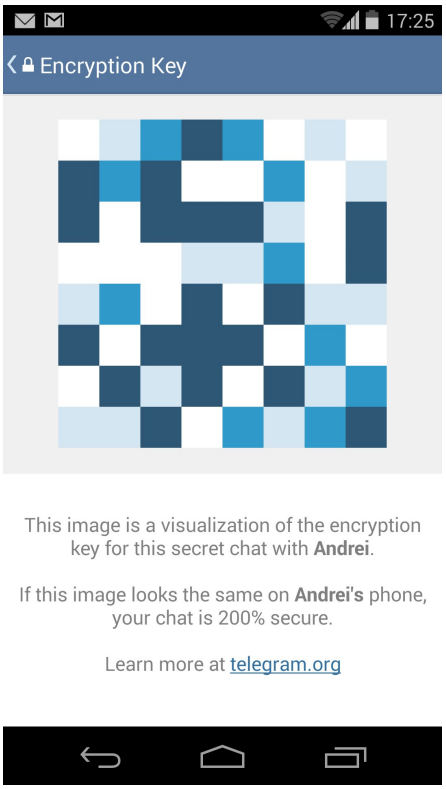

Figure 3.2: Telegram Fingerprint Visualization [2].

concatenated to the encrypted message so it could be checked for modifications after the decryption. Figure 3.3 shows encryption overview.

Telegram's secret chats support perfect forward secrecy by doing a re-key according to MTProto at every 100 messages or a week, whichever is reached first.

3.2.6 Local Storage

Local storage format and whether it should be encrypted or not, is not defined in Telegram protocol. That decision is left to application developers. Official Android application does not use disk encryption [48].

Cloud Storage: Normal messages and all the media transfered within normal

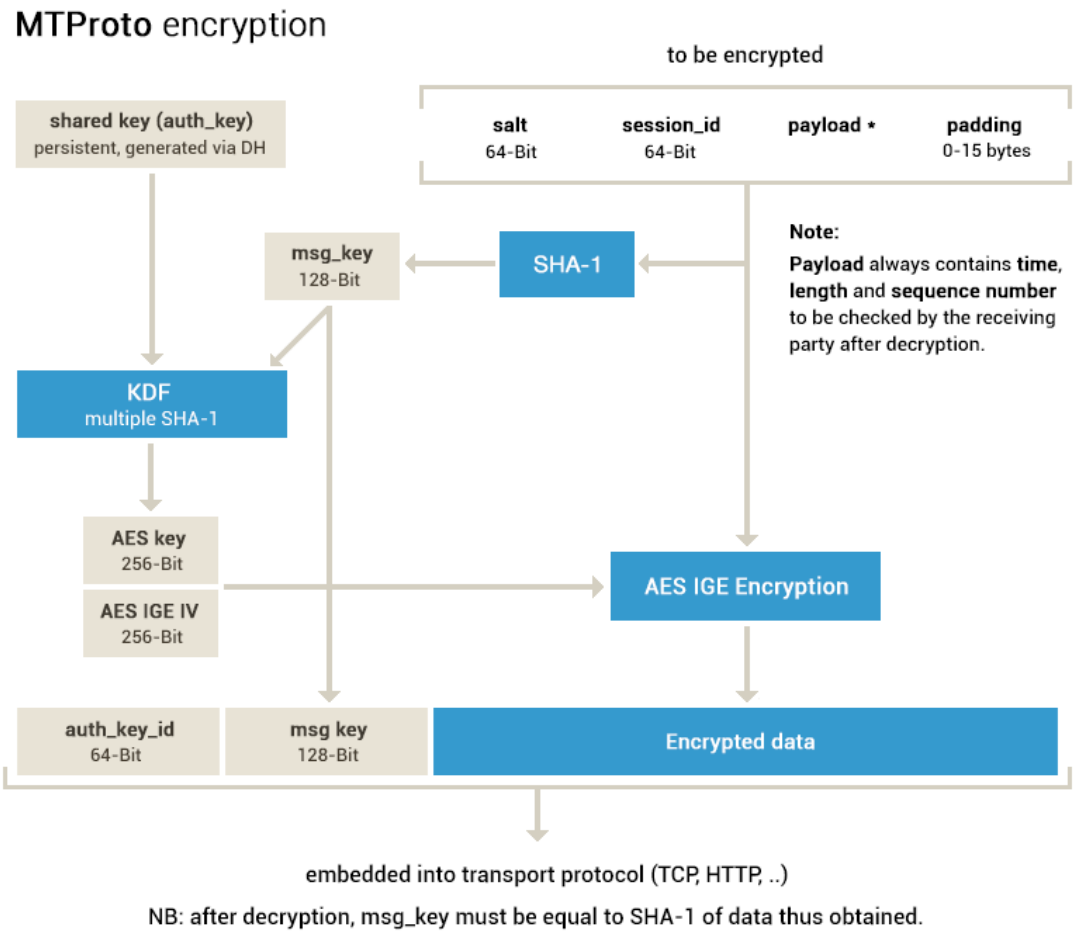

Figure 3.3: Telegram MTProto Mobile Protocol [3].

messages are stored at Telegram's servers to provide cloud message synchronizing. If users are not comfortable with this level of security (Telegram's being able to access all of the messages), they have to use secret chat. Messages which are transmitted using secret chat are end-to-end encrypted.

3.2.7 Message History Transport

Normal chats are automatically saved at the Telegram servers and when the account is moved to another device, it is synchronized with that device. Although,

the synchronization process is secured against third party attackers, normal chat data is available to Telegram.

Secret chats are not saved at Telegram servers so there is no synchronization option for them. Also, Telegram does not provide a way to transfer them to another device.

3.2.8 Public Key Change

Normal chats continue where they are left even the keys change. This is expected because with normal chat, the server is already trusted. However, when a secret chat message comes with a different public key, it is regarded as a different chat session. A new chat session is started when a contact reinstalls the application. Then the user can decide whether to trust this change. However, it does not specify that the keys are changed which can be hard for users to consider man-in-the-middle attack possibility.

3.2.9 Telegram Crypto Contest

Telegram holds cryptography contests about every year. So far, two contests have been organized. First one promised $200.000 to anyone who can "break Telegram". This contest received many negative feedbacks [49, 50, 51] from the community because of insufficient attack surface - no known-plaintext attack (KPA), chosen-plaintext attack (CPA) and chosen-ciphertext attack (CCA). No one could break it within the contest's rules, however, someone with nickname "x7mz" found a vulnerability in Telegram's protocol and has been awarded $100.000 [52]. The vulnerability is occurred because of modifying Diffie-Hellman to add "more security" to it. At equation 3.1 the vulnerable key generation is shown.

$$key = \left((g^a)^b \mod dh\_prime\right) \text{ xor } nonce \tag{3.1}$$

Here *nonce* is created by the server and then sent to clients. An evil server can easily create different *nonce*s for each client and carry out a man-in-the-middle attack. Telegram's reason to use an extra *nonce* parameter to Diffie-Hellman is that they did not trust the PRNG of platform citing a vulnerability [53] of Android.

With the second contest, both prize and attack surface is increased. The promised prize was $300.000 and active attacks are included within the hallenge [54]. This challenge also did not bring out a winner [55].

3.2.10 Telegram Security Issues

An audit [56] on Telegram's source code by Jakob Jakobsen and Claudio Orlandi shows that Telegram does not provide indistinguishability under chosen-ciphertext attack (INDCCA) [57]. This means an attacker can turn any ciphertext into a different ciphertext which decrypts to the same message. It is stressed that this is a theoretical attack and does not allow full plaintext-recovery attack.

Ola Flisbäck's "Stalking anyone on Telegram" post [58] explains how to *stalk* anyone's online status if you have their number and they did not turned off "last seen" setting explicitly. By default, Telegram sends notifications to all of the contacts whenever the application becomes *foreground* or not. Thus, contacts know when the application window is opened or closed which may lead up to "who is chatting with who" information if two sides are the attacker's contacts. Another problem here is that, this meta-data can be visible if the target is added as contact. Adding contacts does not require validation from the other party, so if you know someone's phone number, you can "stalk" his/her Telegram activity provided that he/she did not turn off "last seen" setting.

3.3 Threema

Threema is a Swiss proprietary Android, iOS and Windows application and it is sold for about $1.99 [59, 60, 61, 62]. It has been especially popular after WhatsApp's acquisition by Facebook [63]. Its Android application has a download count between 1.000.000 and 5.000.000 [60]. Although Threema is not open source, they have published a white paper [4] to explain its cryptography internals. Most of these analyses benefit from it.

Threema provides end-to-end encryption. It has perfect forward secrecy support between client and server but not between clients [64]. HTTPS or a custom lightweight protocol is used to communicate with Threema servers and its certificates are pinned in the application against certificate authority compromises.

3.3.1 Registration

At application's first launch, clients randomly generate a long term key pair (LTK) using Curve25519. To increase its randomness, user input is requested. User moves his finger on the screen and these movements' time and location data is collected to provide extra entropy. After the key pair is generated, its public key is sent to the server. Server associates it with randomly generated 8 byte username. Apart from this unique username, users can also specify a public nick name. Optionally, users can link their account with their e-mail addresses and phone numbers. E-mail and phone number are separately verified to associate them with the account.

3.3.2 Authenticating the Server

Threema uses three different servers: chat server, directory server and media server. Chat server uses a custom protocol over TCP. Others use HTTPS.

Chat Server: All of the message transport (incoming & outgoing) is managed by this server. It uses a custom protocol so message headers and connection setup round-trips are minimum. Users are authenticated with their public keys. Its protocol provides Perfect Forward Secrecy (Application restart triggers re-keying).

Directory Server: This server is used when registering and obtaining the public keys of other users. TLS is used when connecting to directory server.

Media Server: When sending media files, this server is used. First, file is encrypted with a randomly generated symmetric key, then it is uploaded to the server over HTTPS. The encryption key is sent to the recipient using end-to-end encrypted message. Finally recipient can download the file and decrypt it with the key. After the download, file is deleted from the server. This server also uses TLS to secure the transport layer.

According to the a FAQ question [65] all servers are protected against man-in-the-middle (MitM) attacks by using embedded public keys in application code. Threema does not disclose its servers' domain names but we have seen that it connects to a server at domain "*api.threema.ch*". This server uses a certificate which is signed by their own CA (not trusted globally) and this custom CA is probably embedded in the application code (certificate pinning), so the application trusts it. A security assessment report [66] by Hristo Dimitrov et. al. says they had tried MitM attack by using different certificates and couldn't succeed which confirms their usage of certificate pinning.

It is noted that Windows Phone 8.0 does not support certificate pinning, so regular certificates are used on Windows Phone.

3.3.3 Authenticating the Client

After username and long term key pairs (LTK) is paired at the server during the registration phase, server can authenticate the clients using LTKs. Client-to-server communication uses Short Term Key pairs (STK) which are created using LTK of the device. These STKs are refreshed at each application startup or 7 days and only stored in RAM. Thus perfect forward secrecy is provided on the network connection level not on end-to-end level.

3.3.4 Client-to-Client Authentication

When the application is first installed, users are presented with the option of synchronizing their contact list with Threema servers. After synchronization, the contacts who have installed Threema is imported into the application's contact list. Users can also add contacts manually by entering their Threema ID (8 characters). When messaging with contacts, Threema servers authenticate users by using their long term key pairs and messages are delivered accordingly. However, Threema users can verify the person they are messaging to, by looking at "three dots" verification level. Every contact is displayed with a three dot verification level, from lowest security to highest:

- One red dot means that either ID and public key is received from the server because the user received a message from that contact or the contact is manually added. No authentication is done on the credentials of the contact.
- Two orange dots means that the contact's email or phone number have a match in user's contact list. Contact's e-mail address and phone number are verified by the server and the contact information is retrieved from the server, so if the server is trusted, the contact can be trusted.
- Three green dots means that an in-person QR code verification is done to verify the contact. Even if the server and network are not trusted,

user can be sure that he/she is messaging with the correct person.

3.3.5 Message Encryption & Perfect Forward Secrecy

Threema uses NaCl Cryptography Library [67] for its cryptography functions. NaCI library is based on public key authenticated encryption. It provides some high level functions to easily encrypt and decrypt messages. At the core level, Curve25519 elliptic curve Diffie-Hellman (ECDH), Salsa20 stream cipher and Poly1305 message authentication codes are used.

Shared secret is created with ECDH (Curve25519) and then hashed with HSalsa20. While encrypting messages, random amount of padding is added to each message then shared secret and random nonce are fed into XSalsa20 stream cipher to generate ciphertext. 128 bit message authentication code (MAC) is generated using Poly1305 on ciphertext. And finally MAC, ciphertext and random nonce are sent. Figure 3.4 shows how the encryption takes place. As we can see from the figure, no ephemeral keys are used for encryption; keys are created using Diffie-Hellman and directly used. Thus perfect forward secrecy is not supported. This means if an attacker get access to private key of one side and message exchange history, he can decrypt all of the messages.

Threema is not an open source application, so you cannot be sure if it works as the way described. However, a "Validation Logging" feature [68] is included within the application. When enabled, it logs raw encrypted outgoing and incoming messages to a file. This file can be retrieved via e-mail for inspection. Using the sender's public key and the recipient's private key, encrypted messages in the file can be decrypted. Required programs are served on Threema's official website [68]. Still, one cannot be sure if the logged messages and sent messages match and no other dangerous code exists.

The encryption approach of Threema does not support perfect forward secrecy which is probably the biggest missing security feature of the application.

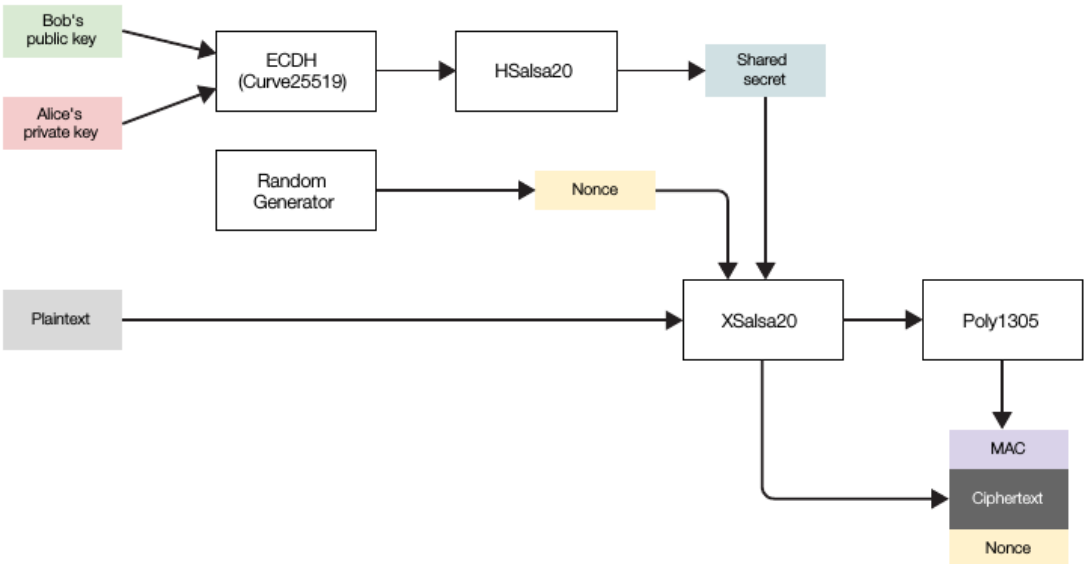

Figure 3.4: Threema Message Encryption [4].

3.3.6 Local Storage

Encryption of the locally stored data is different for each platform. On iOS, Threema uses a native disk encryption feature: iOS Data Protection. The key for encrypting the data is derived from device's PIN. Commonly used 4 digit PINs may not be optimal against brute-force attacks, users should use more complex device pass codes to get the most of local data encryption.

On Android, local data encryption is handled within the application. At first start, a randomly generated key is used for encryption of messages and long term key pair's (LTK) private key. Optionally, this randomly generated key can be protected by a *master key*. To do that, users must go to the settings screen and enable master key option and then set a passphrase. Of course if master key is not used, no real local data encryption is used on Threema's side.

On Windows Phone, SQL Server Compact Edition is used and database is protected with a password which is derived from the LTK's private key. Optionally, this password can be protected with a user passphrase if master password is set.

If an externally provided password (i.e. master password) is not used, Android and Windows Phone local encryption does not provide any real security. Because the decryption passwords to the databases are already on the disk.

3.3.7 Message History Transport

Threema allows manual backup & restore of the application data. Media files can also be included in the backups. Backups are protected by a password of user's choosing before the backup process started. Cryptographic details of the backup process is not explained in the Threema Cryptography Whitepaper [4]. Backup data is zipped and stored on the device. To restore the data, backup file must be moved to the new device. Upon a fresh install of the application, there is an import backup data option, using that option the backup data can be restored. On iOS, platform's default backup option (iCloud) is used. On Windows Phone, the data is backed up to OneDrive [69].

3.3.8 Public Key Change

When a user reinstalls the application, his/her Threema ID changes. Even the same e-mail and phone number is used, the new user is considered a new user and displayed as another user in other users' contact list. If the user has been verified before, new user comes 'unverified'.

3.3.9 Group Chat

Group messaging is handled by sending the message to all of the members individually. There is no direct server involvement in group creation and server does not know which groups exist and which groups have what members. Although server does not handle group management it can easily discover group members by following message delivery patterns, even with small groups. Suppose Alice,

Bob, Carol and David are in the same group. When Alice sends a message to the group, it is sent to three users as different encrypted messages. Since the three messages' send times are very close, server can easily assume they are in a group. Respective messages would further increase the chance of their being in the same group for server. Especially if other messages came from other group members.

When sending media files, to save bandwidth, files are encrypted with a randomly generated temporary symmetric key and encrypted file is sent to the media server. Then encryption key is sent to group members via already established end-to-end encrypted channel so they can download and decrypt the file by using the key.

3.4 Signal Private Messenger

Signal Private Messenger is an open source secure messaging application which is developed by Open WhisperSystems. It was previously named "TextSecure", later it is combined with voice call application "RedPhone" and renamed to Signal. Voice calls part of the application is out of the scope of this thesis, so we only analyzed its text messaging features.

Signal's Android application has a download count between 1.000.000 and 5.000.000 [70]. Also, it has been added to open source Android based operating system CyanogenMod [71] as its default messaging application [72].

Signal has Android and iOS applications: *Signal Private Messenger* [70] for Android devices and *Signal - Private Messenger* [73] for iOS devices.

Signal provides end-to-end encryption and perfect forward secrecy. Encryption keys are changed per message basis using Axolotl [33] protocol. TLS is used for data channel and server's certificate is pinned in the application.

3.4.1 Registration

After Signal is installed, the device needs to be registered to the server. The first step is verifying the phone number. A verification code is requested from the server for the phone number. The server generates a random number in the range of 10000 - 99999 and sends that number via SMS. When device receives that challenge SMS, the code is confirmed by automatically being sent to the server. While confirming the phone number, device credentials are created and sent to the server as Basic HTTP Authorization [74]. These first usage of HTTP Authorization credentials creates and associates supplied credentials with the device. Thus, device can use these credentials for future requests to authenticate itself.

For data channel message delivery, Signal uses Google Cloud Messaging (GCM) [75]. GCM is a push notification service which allows sending small data messages (up to 4 kb) from server to devices.

100 *PreKeys* are generated at the registration and sent to the server upon successful completion of registration. These PreKeys are later used to create shared secrets even when the other party is offline.

PreKey: a *PreKey* is a signed key exchange message. PreKeys are used for generating shared secrets. Each PreKey consists of three parts:

- **public key**: 32 byte randomly generated Curve25519
- **identity key**: 32 byte Curve25519 public key
- **key id**: 24 bit unique key identifier

password: 16 byte randomly generated password. This password is used in HTTP Authentication along with the phone number.

signalingKey: *signalingKey* is a 52 byte randomly generated key to encrypt GCM push messages. The push messages which are sent via GCM are encrypted

so GCM servers cannot access the message meta-data. signalingKey is generated at the registration phase and sent to server while confirming the phone number. It consists of 32 byte AES key and a 20 byte HMAC-SHA1 MAC key.

3.4.2 Authenticating the Server

Signal has three main servers: one for text messaging and two for voice calls. All of these servers use TLS. The servers' certificates are signed by their own certificate authorities but the certificates are pinned in the application as BKS (Bouncy Castle [76]) '*trust store*'s. There are two different trust stores: one for voice servers and one for text messaging. All connections are verified using these trust stores. Thus evil or compromised CAs cannot perform man-in-the-middle attacks.

3.4.3 Authenticating the Client

Each request to the server includes an HTTP Authorization header except the very first API call which is a verification code request [77] call. A Basic Access Authentication is used with phone number and a randomly generated password is used. The password is 16 byte long and generated at the registration phase (see Section 3.4.1). This password is used to authenticate the device after registration is completed. HTTP Authorization header is used as:

```
Authorization: Basic Base64({number}:{password})
```

3.4.4 Client-to-Client Authentication

Each user has a unique identity key. When a user wants to send a message to another user, both users' identity keys are used to generate an ephemeral shared

key. Other user's identity key is retrieved from Signal server. After the handshake is completed, users can compare their fingerprints to be sure of that no man-in-the-middle attack is occurred. After the first message exchange, users save each other's identity keys. If other user's identity key changes, user is warned about this change.

In traditional shared key generation protocols, both parties need to be online to generate an ephemeral shared secret. However, this introduces some restrictions for asynchronous messaging systems. For example, in order to send a message to someone, he/she needs to be online. Otherwise you cannot create an ephemeral shared secret and send message. Some protocols (like PGP [78]) solve this offline message problem by not creating ephemeral shared secret (instead recipient's public key is used), thus not providing perfect forward secrecy.

To solve perfect forward secrecy problem with asynchronous messaging, Signal introduces a *PreKey* approach: client generates 100 signed key exchange messages at registration time (see Section 3.4.1) and sends these PreKeys to the server. These PreKeys are stored at the server. Usage of the PreKeys while sending a message to an offline recipient is defined in [79] and as follows:

1. Request the destination's next PreKey from the server
2. Generate its own part of the key exchange message
3. Using PreKey and own part of the key exchange, calculate a shared secret
4. Encrypt the message using shared secret
5. Concatenate PreKey id, key exchange message, the ciphertext and send it to the destination client

The recipient gets all the data he needs to decrypt the message in a single push when he become online. This approach provides perfect forward secrecy for 100 messages from different senders. If all of the 100 PreKeys are exhausted, then one last resort key is used for all of the other messages. In this case, perfect forward secrecy would not be provided for the first messages of conversations.

3.4.5 Message Encryption & Perfect Forward Secrecy

Signal uses no header keys variation of Axolotl Ratchet [80]. Axolotl Ratchet [33] is a joint work of Trevor Perrin and Moxie Marlinspike. It aims to provide improvements over Diffie-Hellman ratchet of Off-the-Record Communication [29]. Axolotl Ratchet combines forward secrecy and *future secrecy*. Forward secrecy is ensuring compromisation of a session key does not compromise previous sessions' security. *Future secrecy* is defined as healing leaked keys by introducing new Diffie-Hellman ratchet keys [33]. To introduce new keys a *send/receive* pattern is used instead of OTR's *advertise/receive/send*. This two step ratchet simplifies message struct by removing key identifiers as well as using one step less to derive new key. Axolotl protocol also has the ability of protecting messaging meta-data (such as identities or sequence numbers), however Signal's *no header keys variation* does not use this feature.

To begin exchanging messages, parties needs to have shared Root Key (RK) and Chain Key (CK). Generation of these shared keys are described in Figure 3.5. Using elliptic curve Diffie-Hellman (ECDH) protocol, users exchange g^A, g^a and g^B, g^b to calculate three shared secrets: g^{aB}, g^{Ab}, and g^{ab}. Then these three shared secrets are combined as one and used as seed of HMAC-based Extract-and-Expand Key Derivation Function (HKDF) [81]. For HKDF *salt*, zero bytes are used and "WhisperText" is used as *info*. HKDF produces 64 byte output, its first 32 byte is used for Root Key (RK), and the second 32 byte is used for Chain Key (CK).

Message Key (MK): This key is used to encrypt messages.
Chain Key (CK): Chain key is hash iterated to derive Message Key when one party is sending continuous messages.
RootKey (RK): Root key is used to derive new Root Key and Chain Key. It is updated at every ratchet using elliptic curve Diffie-Hellman (ECDH)

The overall picture from Alice's perspective can be seen in Figure 3.6

```
A: Long-term identity key of user A
B: Long-term identity key of user B

a: Ephemeral key of A
b: Ephemeral key of B

if (user A) {
  ECDHE(b, A)
  ECDHE(B, a)
  ECDHE(b, a)
} else {
  ECDHE(A, b)
  ECDHE(a, B)
  ECDHE(a, b)
}
```

Figure 3.5: Shared Key Generation Pseudocode.

Each message is encrypted using message keys (MK). A message key is 32 byte long: 16 byte encryption key and 16 byte MAC key. Prior to protocol version 3, AES CTR mode with no padding is used for encryption, and after that the cipher is switched to AES CBC mode with PKCS5Padding. For MAC HMAC-SHA256 is used and the resulting MAC output is truncated to 8 bytes.

3.4.6 Local Storage

Signal allows optional local encryption of the messages. A master key is derived from a password created by user at the registration. It is used to encrypt and decrypt locally stored messages. An encrypt-then-authenticate approach is used with AES-CBC and HMAC-SHA1.

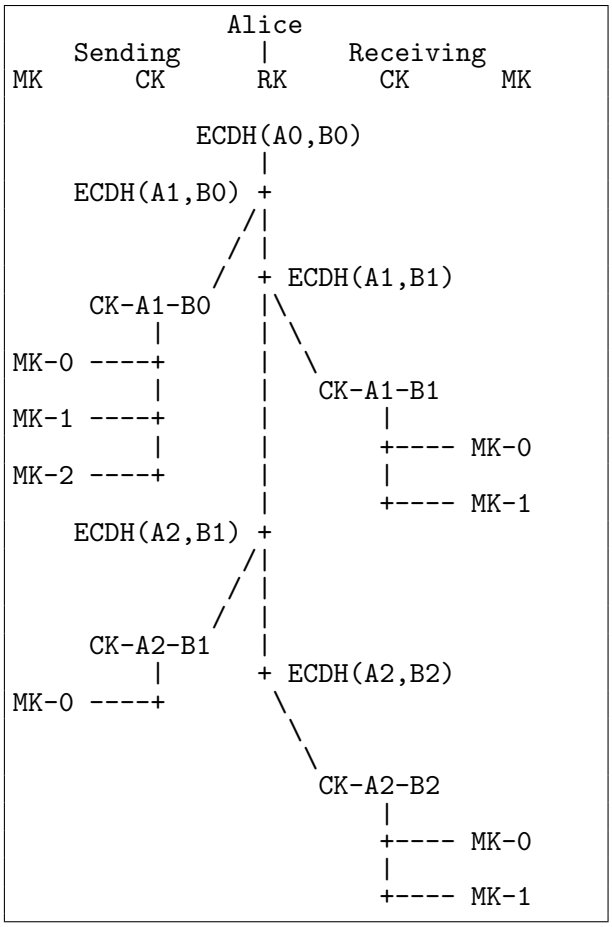

Figure 3.6: Signal Message Exchange. When one side is sending consecutive messages, MKs are hash iterated and when a ratchet is completed new keys are generated with ECDH [5].

3.4.7 Message History Transport

Signal allows plaintext export of the messages. Encrypted export was available but temporarily removed due to a bug [82]. Now importing of the plaintext and previously created encrypted backups are available.

3.4.8 Public Key Change

When a contact sends a message with a different public key than previous messages, then user is notified about this change with a special message balloon. When clicked, it is stated that someone may be intercepting the conversation or the other party is reinstalled the application. User can either accept the new key or cancel the dialog.

3.4.9 SMS Transport

In addition to sending messages over the Internet, Signal could use SMS channel to send encrypted messages [80]. However, in 2015 SMS support is dropped [83] with version 2.7. There is a fork [84] of Signal which continues SMS support.

When sending SMS messages, if the other party also uses Signal, SMS messages were sent encrypted, otherwise sent without any encryption as a normal SMS message. If encryption is used, message is divided into parts to fit into SMS message sizes. There are 60 characters available for the first part, and 155 for other parts. After the message is split into parts, they are sent one by one like normal SMS messages. When the recipient has obtained all of the message parts, they are combined and then decrypted. The recipient sees it as one message.

3.5 Summary

We have analyzed some of the most actively used end-to-end encrypted messaging applications and their protocols. We have inspected their registration and authentication schemes, encryption methods and how they mitigate common attack points. They all try to provide a secure messaging environment while having a good user experience. Because of that, not all applications support all the possible security measures; they choose what they can offer within their usability cases. We present how much security each application offers, what are their advantages,

disadvantages and possible weaknesses.

Cryptocat can be used for iPhones and desktop browsers but it does not have an Android application. Its source code on GitHub is not updated frequently [85] - it has been almost a year without any commits. It is more of a chat client rather than messaging application - you cannot send offline messages to your contacts. Using Cryptocat may be better suited for one time only use. Since public/private key pairs aren't persistent, it requires verification for every chat and this can be cumbersome if long period usage is desired.

Telegram is probably the most actively used application of all. However, we cannot say it is the most secure one. Firstly, it defaults to "normal chats" which are not end-to-end encrypted and the message data is stored on Telegram servers. Secondly it uses a highly custom cryptography protocol which is almost never a good thing. Although we haven't seen it completely broken, theoretical attacks [56] exists and we saw its custom cryptography failed once (see Section 3.2.9). Telegram can be used because of its good UI, large user base and somewhat OK security but there are more secure alternatives to it.

Although Threema is not open source, its security protocol is well documented and looks good. Probably the main disadvantage of its security protocol is that it does not provide PFS between clients. It also has the disadvantage of being a paid application but its usability and security protocol design is makes it a fair choice.

Signal's security protocol design is probably the most advanced of all. It has PFS with ephemeral key update at each message and supports asynchronous messaging from the start of conversation. WhatsApp started working with Signal's author "Moxie Marlinspike" to bring E2E encryption [86]. At the end, Signal is an open source application with a nice user interface and have strong security design. It could easily be de facto secure messaging application.

Apart from the applications we have analyzed, there are other notable applications like Wickr [87], surespot [88] and ChatSecure [31]. They all provide

end-to-end encryption, Wickr and ChatSecure also provides perfect forward secrecy (PFS). surespot has a download count between 100.000-500.000 [89] on Google Play, whereas Wickr and ChatSecure both has been downloaded between 500.000-1.000.000 [90, 91]. Wickr supports video verification for client-to-client authentication which is a nice and practical way for users of all levels of security experience to verify the other party.

To get a quick security overview of other messaging applications, Electronic Frontier Foundation (EFF)'s *Secure Messaging Scorecard* [92] can be used. It is based on 7 item checklist: "Encrypted in transit?", "Encrypted so the provider can't read it?", "Can you verify contacts' identities?", "Are past comms secure if your keys are stolen?", "Is the code open to independent review?", "Is security design properly documented?" and "Has there been any recent code audit?".

Chapter 4

Our Design

After evaluating existing end-to-end encrypted messaging applications, we propose our solution as an Android application. Our solution tries to get the best of existing applications and protocols while maintaining same usability levels as common "unsecure" mobile messaging applications. Our solution offers an enterprise-friendly secure messaging application. We want to support messaging for both end users and enterprise users. Enterprise users may have requirements for on-premise setup for servers, in-house usage and using existing authentication mechanisms like Lightweight Directory Access Protocol (LDAP).

While designing our solution, we tried to make the application as secure as possible without compromising user experience. We determined the security requirements as:

1. Use End-to-End Encryption: No one but users should access the message or its meta-data. This includes application servers, Internet service providers and platform's push message servers.

2. Implement PFS: In case of a possible key compromise, past messages shouldn't be accessible.

3. Use local encryption to save message history: Messages should only be

accessible after authenticating the user and there shouldn't be any sensitive data when the device is turned off.

4. Use a password for accessing to the application: The sensitive content should be locked when the application is not in use and by providing a password, message reading and sending should be possible. For usability, this shouldn't be same with the credentials used when registering (Do not make users frequently type e-mail and password).

5. Use a secure random generation: Try to harden the security of default random number generation.

6. Be resistant against brute-force attacks: When encrypting message data and when using key derivation functions do not use insufficient length keys and obsolete algorithms. Also, do not use unnecessarily long keys because of mobile devices' relatively low power.

Apart from security related requirements, we also have system design requirements. We have designed the system so that these requirements don't weaken the security of the application. The requirements are listed as:

1. Have single application code to support both a central server and on-premise servers.

2. Make it easy to use and configure for different security requirements. Security shouldn't be a burden for average users.

3. Make it as scalable as possible. This is mostly a server-side architectural problem but some security protocols affect the scalability of the system. Security should not limit the system's scalability.

4. Support both LDAP and a username-password authentication on server-side and thus, on device.

In the light of our security and general system requirements, we discussed our design decisions, carried out tests to help our design, presented and interpreted tests results and explained the application's protocol details.

4.1 Registration

To use a messaging system, a user needs to be associated with some unique attribute which he can verify. This is usually a phone number for common messaging systems. Phone numbers' ownership can easily be verified by sending a token via SMS. Since using phone numbers is an already established way of communication for normal phone calls or SMSes, it is easy to integrate. However, some users may prefer to use a separate identifier to remain anonymous, in that case, e-mails or randomly assigned identification numbers can be used.

If the messaging system is used within an enterprise, their already existing authentication methods can be supported. It is valuable to provide single sign-on experience or at least allowing to use same credentials for enterprise's services. As a standardized and widely used method, we have added Lightweight Directory Access Protocol (LDAP) authentication support for our system. As an alternative, internally creating users with username & password credentials are provided.

When a user is registering, the username and password is sent to the server. Then the server validates them according to a pre-defined method (LDAP or internally created users). If they are valid, server sends an authenticated JSON Web Token (JWT) to be used at other requests. Client stores the JWT data and sends it at every request's header. After client application receives the token, registration phase is completed.

We haven't integrated phone number only registration at the moment because in most enterprises it is not desired that anyone with a phone number can register and use this internal service. Although if the need arises, changing authentication

method can be implemented modularly without breaking current secure communication structure.

4.2 Authenticating the Server

The very first step of end-to-end messaging is authenticating the server. Because clients cannot talk to each other directly, all messages need to go over trusted server(s). So how can we decide the server we are trying to connect is the real server? With SSL/TLS and X.509 Public Key Infrastructure, we can be sure that we are connecting to a trusted server as long as we are fully trusting all certificate authorities. However, we have seen [14, 15, 16] this approach can be problematic.

Today many security-oriented applications use certificate pinning which seems the most complete solution to authenticate the server that we are connecting. Martin Georgiev's paper [93] shows us how dangerous it can be not to use certificate pinning on mobile applications. With certificate pinning, trusted application servers are defined in the application code, so no other server is trusted whether they have valid SSL certificates or not. Certificate pinning moves the trust from certificate authorities to application code. However, according to its implementation there may be a need to update the application if the application server's certificate expires or needs to be changed.

Our application design needs to support both using main server and on-premise servers. To do that, we check the domain name of the e-mail address provided as username; if it is equal to our main server's domain name then we say it is used as software as a service (SAAS) and use the pinned certificate for the main server. However, on-premise servers and their certificates are not known beforehand, so we cannot pin them in the application. Rather we have used a trust on first use (TOFU) method for them; if an on-premise server is used, we first check whether this is the first attempt to connect the server, if so we trust its certificate (provided that it is also a valid SSL certificate) and save the certificate in local database. Consecutive requests checks if the certificate is consistent with

the stored certificate. This process' algorithm is shown at Algorithm 1.

Algorithm 1 Certificate Checking

```
1: if server_domain = main_server then
2:     check the certificate against embedded certificate
3: else
4:     if is first connection then
5:         if the certificate is valid then
6:             save the certificate
7:             continue with connection
8:         else
9:             abort the connection
10:            alert the user about possible attack
11:        end if
12:    else
13:        check the certificate against saved certificate
14:    end if
15: end if
```

4.3 Authenticating the Client

Clients are authenticated using username and password. Username-password's validity is checked by server using either LDAP or an internally generated list of valid username-passwords. After this initial validation, an authenticated JSON Web Token (JWT) [94] is generated for the client and send to the client. This token has an expiration time and authenticated by the server's secret key. So when the client resends this token, server can be sure of that the client has been authenticated.

To make this operation more user-friendly, expiration time can be made very long (years) so that clients wouldn't need to re-login using long usernames and passwords. However, this token should be treated as private data because it can directly login clients to the server.

JSON Web Token: JSON Web Token (JWT) is an RFC standard [94] which is used to securely identify a client connecting to server. It plays a similar role to HTTP web cookies. JWT is defined as JSON and it is authenticated and/or

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

Figure 4.1: JWT Signature Generation.

```
{
  "alg": "HS256",
  "typ": "JWT"
}
{
  "username": "burak",
  "id": 123,
  "exp": 1449921352,
  "roles": ["admin", "developer"]
}
```

Figure 4.2: JWT Header and Payload Example.

encrypted. JSON representation of some data is authenticated using a secret by an authority (could be anyone) then it can be distributed to clients. When a token is retrieved from a client, its signature can be checked against authority and we can be sure about its authenticity.

A JWT consists of three parts: Header, payload and signature. Header part includes algorithm and type parameters and *base64Url* encoded. Payload part is where the data is defined. There are some reserved fields for issuer, expiration time, subject etc. This payload is also *base64Url* encoded. Last part is the signature part, this part is generated according to selected algorithm at header part. An example JWT which uses HMAC SHA256 can be seen in Figures 4.1 and 4.2.

JWT token is added as HTTP authorization header with the format:

```
Authorization: Bearer <token>
```

When server gets the JWT token, it checks whether the signature is valid and it is expired or not. If both checks pass, server can trust that the client has been authenticated by a trusted server (most probably itself, but in a multi-server environment it may be other servers).

Using JWT at client side requires no additional library, clients just need to save and add it to HTTP header. At server side there are many tools available to create and check validity.

JWT can be used by different domains and different servers, no session ids are stored and database is not queried with each request so it is more scalable than traditional cookie approach. For these architectural reasons we chose JWT over cookies.

4.4 Client-to-Client Authentication

With end-to-end encrypted messaging, clients send encrypted data to each other which are only decrypted by the clients themselves. However, there is still a risk of impersonation or man-in-the-middle (MitM) attacks. Somehow clients need to be sure they are messaging with the correct person.

Clients need to be able to verify each other and be sure of the security of conversations without trusting the server, GSM operators, Internet providers or even certificate authorities. Providing a fully secure model requires many beforehand instructions to start a conversation. This makes it harder to deploy such a system. This brings us to security-usability trade-off. We think the application must provide as much as security possible without requiring additional confusing steps (comparing to their 'unsecure' commodity alternatives) and also provide options to verify the security status when needed. For this reason, we start the client-to-client authentication with a trust on first use (TOFU) model. Initially, clients save their contacts' publics keys and trust these keys. If these keys change, the user is alerted about the change in a user-friendly way. In the alert message,

user is presented with possible scenarios: a MitM attack, application reinstall or phone change. And they are encouraged to contact to the person if they are in doubt.

This TOFU approach allows server to send evil public keys to clients. Because of that, we allow users to compare their public keys in person. This way, they can be sure of that they received correct initial keys. If keys are compared once, they can continue messaging securely as long as keys don't change. Even then users can compare new keys anytime they desire.

There are different ways of displaying and verifying public keys. Usually their fingerprint (a hash output of them) is used instead of comparing the whole public keys. There are different ways to display these fingerprints: Hexadecimal strings, Base64 encoding, QR codes, encoding to words, other custom visualizations etc. We have used hexadecimal strings because of both implementation simplicity and no fingerprint data is lost or long word lists are generated. QR codes look like good way to quickly compare some random-looking data but they require additional applications to use, and if users don't already have them installed, it may be cumbersome for them to install new application, instead they may skip the whole process.

4.5 Message Encryption & Perfect Forward Secrecy

Perfect forward secrecy (PFS) is a protection against the compromise of past communication data. Basically, if the same encryption key is always used, the compromise of it would result in possibility of decrypting all of the previous communication data. A secure communication system should support PFS so that when a long term private key compromise occurs, it should be impossible to decrypt previous communication data. Solutions with PFS gives users the trust of "if you communicate securely now, your past communications cannot be

decrypted later.". A well designed secure messaging application should support PFS so that users won't fear that their messages can be decrypted later.

Our design adopts an OTR-like approach: Both sides sign random numbers and Diffie-Hellman key exchange is done using these numbers (See Figure 4.3), not directly the certificates. This way, each side can authenticate these numbers with the help of signatures on the numbers.

x: random number chosen by A
y: random number chosen by B
A $\rightarrow$ B: $g^x, Sign_A(g^x)$
B $\rightarrow$ A: $g^y, Sign_B(g^y)$
SS=g^{xy}

Figure 4.3: Initial Diffie-Hellman Key Exchange.

To provide PFS, the shared secret should be updated periodically. To update it, one side chooses a different random number, and appends it to the message by encrypting with the current shared secret. Other side receives this message with the next number and sends a random number of his choice. After the second number is received by the first side, new shared secret is generated (See Figure 4.4) and can be used from now on. Until both sides get the new shared secret parameters, previous shared secret is kept. Message counters are also used in the process because message order can be different in some cases and if the order is not satisfied, previous shared secrets may be saved temporarily. However, after some time, they need to be destroyed and the messages which are encrypted with them should be discarded.

x': random number chosen by A
y': random number chosen by B
A $\rightarrow$ B: $Encrypt_{SS}(g^{x'})$
B $\rightarrow$ A: $Encrypt_{SS}(g^{y'})$
SS'=$g^{x'y'}$

Figure 4.4: Re-Keying of the Shared Secret.

The shared secret key is used for encryption and authentication of the message. We need both encryption and authentication to be sure that messages are not readable by anyone and cannot be tampered. We have used an encrypt-then-authenticate approach: AES with CBC is used to encrypt the data and HMAC-SHA256 to authenticate the encrypted content.

According to NIST [95] 128, 192 and 256 bits of AES key implementations are sufficient to protect information up to the SECRET level; 192 or 256 bit key length is required to protect TOP SECRET information. However, Alex Biryukov's paper [96] describes attacks against 192 and 256 bits AES implementations. Although these attacks aren't practical currently, running on mobile devices' relatively low computation power and knowing AES-128 provides enough security, we opted for 128 bits of encryption and authentication keys.

K1: Encryption key
K2: Authentication key
plaintext: Data to be encrypted
IV: Randomly generated initialization vector

encrypted = $Encrypt_{K1}(IV, plaintext)$
MAC = $HMAC_{K2}(IV|encrypted)$
ciphertext = $IV|encrypted|MAC$

Figure 4.5: Message Encryption.

Encryption code is described in Figure 4.5. While encrypting a message, first, a random IV of 128 bit is generated, then this IV is fed into the encryption function along with encryption key and plaintext. Resulting ciphertext and the IV is fed into HMAC function. It is vital to add the IV to the MAC function in order to prevent IV modification attacks. Lastly IV, ciphertext and MAC is concatenated to create the final ciphertext.

Decryption code is described in Figure 4.6. To decrypt, first MAC is calculated for the IV and encrypted data part and then it is compared with the MAC from the ciphertext. If they don't match, process is aborted and user is alerted about

ciphertext = $IV|encrypted|MAC$
MAC' = $HMAC_{K2}(IV|encrypted)$
assert(MAC = MAC')
plaintext = $decrypt_{K1}(IV, encrypted)$

Figure 4.6: Message Decryption.

a possible MitM or corruption. Otherwise, encrypted data is decrypted using the encryption key and IV, thus plaintext is obtained.

4.6 Local Storage

In a typical messaging application, previous messages are saved on a persistent storage (disk) to provide a better context for conversations, keeping message drafts and archiving purposes. Secure messaging should not be different if maximum usability is desired. The problem with storing messages on disk is that, once plaintext data is written on disk, it can be retrieved when the device is turned off or if the device is rooted other applications may read messages easily. To overcome these issues, messages are usually encrypted and then their ciphertext is stored on disk. When the messages are to be read, message data is decrypted then displayed to user.

Some implementations (like an older version of WhatsApp [97]) also store encryption key on the same disk which doesn't introduce a new level of security, just makes it a little harder for attackers to read messages. If an attacker has access to data on disk, he can both read the encrypted messages and encryption key, then decrypt the messages. In a typical Android environment, attacker also has access to application package which can be decompiled and encryption algorithms can be discovered rather easily. Also, we shouldn't rely on security through obscurity from the beginning.

In a secure environment, messages should be stored encrypted and decryption

key should be provided by user to unlock messages. However, it isn't very realistic for users to use and remember 128-256 bit encryption keys. On the other hand if the encryption key is directly used as user's password then it wouldn't be very secure. To address this problems, we generate a pseudorandom key for encrypting messages then this key is encrypted with user's password and then this ciphertext of encryption key is saved on disk. To decrypt, user supplies his/her password which decrypts the encryption key then encrypted messages can be decrypted. These decrypted messages should only reside in RAM, if somehow they are written to disk, we would return to the same problem with plaintext on disk.

Every Android application runs on their own process (only applications signed by same signature can be run on same process [98]) and their data directories have their process' permissions. Thus Android applications cannot access to other application's data unless data is intentionally exported. However, if the device's disk is examined outside of Android platform (i.e., connecting the disk to a computer), the data can be retrieved. This data can be anything from all of the user's messages, contacts, e-mails or any other private information. Full disk encryption may greatly reduce the data leak if the device is lost. However, full disk encryption is not available and enabled on all devices. It also uses same unlock mechanism as screen unlock and it may not be very secure to use same password with frequently used screen unlock mechanism for encrypting disk. Rooted devices may also break Android's security sandbox and access the data while device is powered on - in this case, full disk encryption wouldn't block the attackers. With rooted devices reading device's RAM or other attack strategies may be possible but these are not in the scope of this thesis.

SQLCipher: SQLCipher [99] is a security extension for SQLite which provides using encrypted databases instead of normal plaintext databases. It uses almost same API as normal SQLite, so switching to SQLCipher doesn't require much work. Only fundamental difference is when opening a database file, a password needs to be provided. This password is used when encrypting/decrypting databases. The password is fed into a PBKDF2 function (default iteration count is 64000) and each database is initialized with 16 bytes of random salt. This provides that the same passwords do not produce the same encryption keys. Default

algorithm for encryption is 256-bit AES in CBC mode but this can be changed. SQLCipher encrypts both records and meta-data (table names, column names etc.) which makes it harder to reverse-engineer the application.

The main disadvantage of SQLCipher is that it increases the application's size greatly. We tested the difference with a "hello, world" like application for database. The application opens a database, creates a table and writes one row to the table. This application is built for "release" and optimized using ProGuard [100]. The resulting Android application package (APK) file size is 492 KB. The same application is set to use SQLCipher with language asset and native libraries for different CPU types (arm, arm-v7a, x86). It is also built for "release" and optimized using ProGuard with same parameters. Resulting APK file size was 7.7 MB. SQLCipher has increased the APK size by about 7.3 MB. Generating different application packages for different CPU types and creating a custom language file may provide some reduction on this increase.

Because of SQLCipher's excessive package size increase, we also implemented a method to manually encrypt each critical field of the records, so the APK size would not increase that much. Each sensitive field is encrypted using with local storage encryption key 256-bit AES in CBC mode. Resulting byte array is stored as BLOB data type. This method requires more work and doesn't protect against the plaintext meta-data leak. However, it doesn't increase the APK size that much. The increase would be probably under 1 KB since only wrapper functions for existing AES cipher is added.

Manually encrypting each (or some) field requires more work and produces lower maintainable code base but the increase in APK file is negligible. If network bandwidth or disk size is restricted or competitive advantage of small APK size is desired, manual encryption of fields can be used, otherwise using SQLCipher should be preferred.

```
password: generated by user
encryption_key = scrypt(password)
encrypted_data = encrypt(backup_data, encryption_key)
```

Figure 4.7: Backup Encryption

```
encryption_key = scrypt(password)
backup_data = decrypt(encrypted_data, encryption_key)
```

Figure 4.8: Backup Decryption.

4.7 Message History Transport

It may be desired to support moving message data from one device to others, especially when users buy a new device or reset their devices. This transportation of messages process should not introduce vulnerabilities or weaken the security of the system. We have designed a simple and secure export & import functionality which is based on creating encrypted backup files and importing them to the new environment. During the process, no plaintext data is written on disk.

When exporting the message history data, user is expected to provide a password to encrypt the backup file. This password is fed into a KDF to create encryption key. The data is encrypted with this encryption key using 256 bit AES with CBC mode (See Figure 4.7). It is left to the user to transfer the backup file to the new device. Third party file hosting integrations may also be used to provide an easier transition. To import the messages, user needs to select the encrypted backup file, then enter the password which is used for encrypting the backup. Password is fed into a KDF and then encryption key is obtained. Using the key, file is decrypted (see Figure 4.8) and messages can be restored using the plaintext backup file.

4.8 Public Key Change

For two common cases, reacting to public key changes is required: When a man-in-the-middle (MitM) attack occurs and when the other side reinstalls the application (losing old key pair). We cannot be sure which is happened so user should be informed about the change. It could be via an alert dialog or a separate messaging thread may be opened to distinguish these new messages from the old ones.

We preferred explicitly alerting the user via an alert dialog, otherwise user can continue within the new thread without knowing the MitM attack possibility. A dialog is displayed to the user with options for either trusting or not trusting the new key. If the new key is trusted, the conversation continues; if not, until the key is trusted, no new message is displayed.

4.9 Push Message Servers

Messaging is an asynchronous communication type. It is more like e-mail than chat, because when a user wants to send a message, the receiving end doesn't need to be online. To support this functionality, the receiving end needs to be alerted when a message is sent to him so he can retrieve the message. This can be done in various ways like polling at some time intervals, long polling or using a central messaging server.

Polling at intervals: Devices can continuously ask the server at some time intervals to check if there is any message for them. Although this is easy to implement, there are both user experience and performance problems with this method. First, if a 20 seconds polling time is selected, users will receive messages with an average of 10 seconds and maximum 20 seconds delay. This delays would hurt the messaging experience. Secondly, it is highly battery consuming for mobile devices to create a connection every 20 seconds. And also, having multiple applications using this method would drain the battery quickly. Although this

method can achieve somewhat desired message alerting option, using it is not a very viable option, and sometimes impossible for some platforms (iOS has restrictions about background applications network usage).

Long Polling: With long polling, a connection is established to the server asking for new messages. If there are any new messages, they are sent in the response; if not, the server wouldn't respond until a new message is available or the connection timeouts. With this method users are alerted almost immediately when a new message is available for them. The downside of this method is that when n applications use this method, there would be n active connections which is also very bad for the battery life (worse than polling at intervals).

Central Push Servers: This approach is similar to long polling but instead of every application having a separate connection to their servers, the device opens only one connection to the platform's central push server and receive the messages from it. Application servers send messages through that central server and device delivers messages to corresponding applications. Thus messages are delivered with minimum delay and battery consumption. This approach is the de facto way of delivering alert and messages to mobile applications.

Although using central push servers are the preferred way of sending messages to mobile applications, it is not without compromises, especially from the security perspective. With this method, it is clear that message content is visible to central push servers. So we must approach to it as an unsecure communication channel and we cannot send message content and meta-data in plaintext. When using end-to-end encryption, the message content is already encrypted but its meta-data (message sender etc.) may not be encrypted. If end-to-end-encryption is not used, both message content and meta-data are not encrypted and they are exposed to the central push server. There are two common ways to avoid data exposure to the push message servers:

1. Do not send message content and meta-data, send notification. Only send an alert then the application can fetch the content from application servers using TLS/SSL. Although this is simple and effective, it requires an extra

network step to receive the message.

2. Encrypt the content and meta-data: Encrypting the whole message using a shared secret between device and application server would not reveal any information to the push servers (besides timings of messages).

We have implemented both methods and make the client application capable of processing both cases. It is dynamically configurable from the server. If the server sends a command with *new_command* type; the client requests the new command data from the server, otherwise the command data is read and it is executed according to its type.

4.10 Key Derivation Function

Key derivation functions (KDF) are used to derive keys from passwords or other keys. The input is stretched until an output with desired length is generated. KDFs are useful for converting user supplied passwords to encryption keys, since user supplied passwords cannot be used directly as encryption key if they are not exactly in the same format with expected encryption key format.

We need a KDF for creating a key from user's password which will be used to encrypt the master key. A secure KDF must be hard to reverse and strong against brute-force attacks.

We have analyzed PBKDF2, bcrypt and scrypt and decided on scrypt because of its both space and time requirements make it harder to brute-force. Figure 4.9 shows a comparison between various KDFs. Looking at execution times and brute-force costs, we can say that scrypt performs much better than the alternatives.

scrypt has three parameters: CPU, memory and parallelization cost parameters. CPU cost parameter controls the required CPU power to compute the key, as it increases, CPU cost of key derivation increases. Memory cost parameter

KDF	6 letters	8 letters	8 chars	10 chars	40-char text
DES CRYPT	< $1	< $1	< $1	< $1	< $1
MD5	< $1	< $1	< $1	$1.1k	$1
MD5 CRYPT	< $1	< $1	$130	$1.1M	$1.4k
PBKDF2 (100 ms)	< $1	< $1	$18k	$160M	$200k
bcrypt (95 ms)	< $1	$4	$130k	$1.2B	$1.5M
scrypt (64 ms)	< $1	$150	$4.8M	$43B	$52M
PBKDF2 (5.0 s)	< $1	$29	$920k	$8.3B	$10M
bcrypt (3.0 s)	< $1	$130	$4.3M	$39B	$47M
scrypt (3.8 s)	$900	$610k	$19B	$175T	$210B

Figure 4.9: Comparing scrypt with other KDFs. scrypt requires more resources for brute-force attacks while normal execution times are similar [6].

controls the required memory to compute the key, as it increases, larger memory is required to derive the key. Setting memory cost is important against attacks using application-specific integrated circuits (ASIC). Parallelization parameter controls how many times the function should be run. It is possible to make it "2" and run two functions in parallel. We tried to find optimal parameters according to mobile devices' computation power such that generating the key would not make users wait too much and it is as hard as possible for attackers.

We have seen two commonly used libraries which provide scrypt implementations. One is Spongy Castle [101] - a repackage of well-known cryptography library Bouncy Castle [76] and another one is an open source library wg/scrypt [102] which is hosted on GitHub [40]. wg/scrypt uses Java Native Interface (JNI) for C code implementation on Android. JNI is generally used for better performance at performance critical applications like games or video/-camera applications. Thus when JNI is used, it gives the impression of a faster library/application. However, this is not always the case. To find out which library performs faster, we compared both libraries. We used 4 different values for CPU cost parameter and 5 different values for memory cost parameter on 5 different devices. We have seen Bouncy Castle is approximately two times faster than wg/scrypt library (See Figure 4.10). So we have used Bouncy Castle in our application.

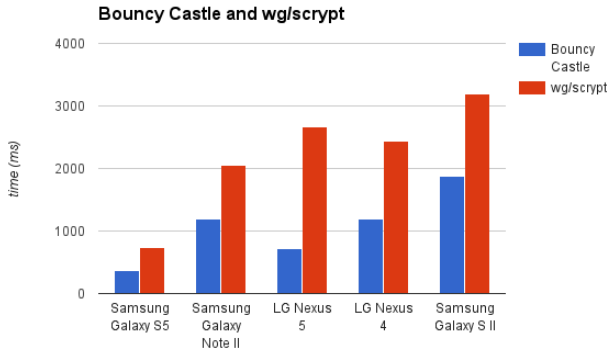

Figure 4.10: scrypt Library Performance Comparison: Bouncy Castle outperforms wg/scrypt.

scrypt provides three parameters for deriving a key. To find the optimal parameters, we have carried out some tests. We tried to keep test values as mobile-friendly as possible and used 1024, 2048, 4096 and 8192 for CPU cost parameter. Further increasing the parameter caused out of memory errors for some devices and also computation time increases over tens of seconds. For memory cost parameter we chose 2, 4, 8, 12, 16 and kept parallelization parameter at 1 because it further complexes the tests and its calculation may differ according to hardware specifications.

KDF is executed when the application is loaded into memory. This can be when the device is powered on or the application is force-closed. In these scenarios, putting a loading indicator and making the user wait a few seconds seems reasonable. We put a usability limit as the operation should be completed within 2 seconds and tried to use largest memory parameter as possible. Our test results are displayed in Figures 4.11 to 4.14. Looking at the execution times we chose *4096* as CPU cost parameter and *12* as memory parameter. Using these parameters, key derivation takes between 0.5 to 3.0 seconds according to device type. And only for our oldest device (2011 model Samsung Galaxy S II), it took over 2 seconds to complete.

As a further improvement, choosing parameters can be decided dynamically

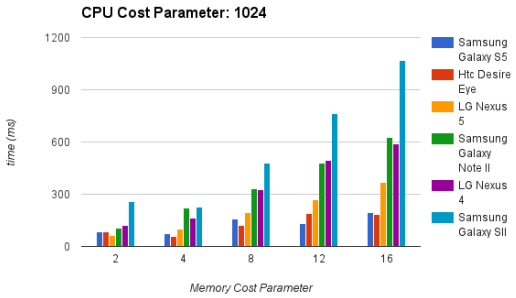

Figure 4.11: CPU Parameter: 1024

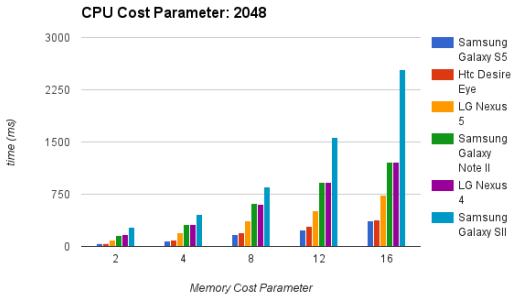

Figure 4.12: CPU Parameter: 2048

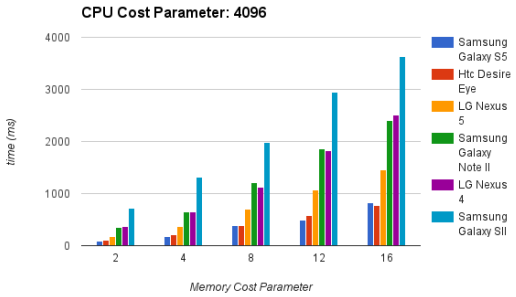

Figure 4.13: CPU Parameter: 4096

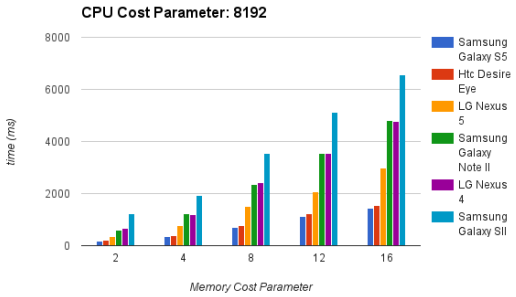

Figure 4.14: CPU Parameter: 8192

according to device's computation power. There are tools to quickly find an approximate device computation power. Facebook's open source device-year-class [103] maps device's computation power to a year as in "this device was a high-end device at that year". Using similar tools, parameter choosing can be made dynamic and as a safety measure some hard limits can also be enforced. Using such a technique makes stronger devices more secure and the application would still operate in a usable zone.

4.11 Creating Secure Random Numbers

Creating secure random numbers is a critical step for the system's security. Most of the time, you can use system's default random number generator. But sometimes they may have weaknesses or do not offer desired security levels. For example, some Android versions (API 16, 17 and 18) have a vulnerability [53]

```
ix: new x coordinate created by touch event
ix: new y coordinate created by touch event

array[i]   = ix,
array[i+1] = iy
array_bits = convertToBits(array)

seed=platform generated number
kdf(array_bits, seed)
```

Figure 4.15: Turning Touch Coordinates to Pseudo Random Byte Array.

which affects creation of good random numbers. Although it has been fixed after API 18, and patches are provided for application developers to mitigate the issue, there is always a chance that platform's random number generation is not good enough.

It may be tempting to use a custom solution but it is almost always a bad idea because doing it right is difficult and you are trying to use a system which is not tested 'in the wild'. Keeping these in mind, we tried to introduce extra randomness by getting input from user onto the platform's default method and not reducing the randomness of platform's default method.

TouchRandom: We have developed a library for getting input from user to help creating random numbers. With users' input we expect higher randomness and a better security feeling from the users' perspective. TouchRandom displays a rectangular box which users can touch and hold various areas to generate arbitrary coordinates and then they are used for creating a random number along with platform's default random number generator. These generated coordinates are fed into a KDF with platform's random number generator then our relatively more secure random number is generated (See Figure 4.15).

A progress bar is added to display the status of fulfilling the required number of points. Bar's percentage is incremented and when it is full, user can finish (or continue to generate even more points). Filling the array takes approximately 5

seconds. An screenshot is displayed as Figure 4.16.

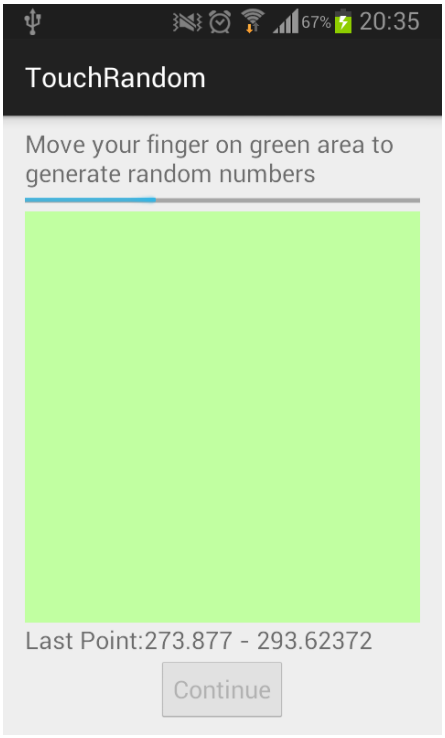

Figure 4.16: TouchRandom screenshot when receiving input from the user. Last recorded coordinates and their precision is displayed.

This method would reduce the risk of vulnerable platform number generators and using a relatively long list of coordinates, its predictability would be very low.

Our test box consists of 430x450 points when using a 480x800 device and a minimum of 200 coordinates are required to move to the next step. This means, for this particular device with 480x800 pixel screen resolution, there are roughly 200.000 different points and we expect users to give at least 200 coordinates input. This should provide plenty of entropy to the randomness. And also including time of the events may further enhance the security of number generation. Besides mathematical calculations, users' random drawing habits may affect expected randomness. For example, with a quick test group we have seen most of the people tends to draw average sized circular paths. Calculation of the randomness

entropy of this method and choosing the best parameters is out of the scope of this thesis and can be carried out as a future work.

4.12 Discussion

Within this chapter, we presented our secure messaging design. We had some security requirements so that users are presented a complete solution for secure messaging. There were basic but critical usability requirements so that using a secure alternative to commodity messaging applications wouldn't be cumbersome. We also had some general design requirements which are mostly about performance, scalability and enterprise support. We had worked on this requirements, tried to explain what should be expected from a secure messaging application and how we have designed our solution to implement them. Security design decisions are discussed in detail and other requirements are mentioned where necessary.

Our design supports end-to-end encryption with perfect forward secrecy (FPS). An OTR-like method is used for shared secret generation and supporting FPS. All connections to servers use Transport Layer Security (TLS). Certificate pinning is used by embedding the server's certificate fingerprint in the application code. This provides protection against compromised CAs, network operators and Internet service providers. We use 256 bit AES with CBC mode for message encryption and HMAC-SHA256 for message authentication. For local disk encryption, we presented two different approaches: Using a third party library (SQLCipher [99]) and manual encryption/decryption of critical fields. scrypt [6] is used as our key derivation function and in addition to platform's random number generation, we also get input from the user, so a relatively more secure random number generation is done.

To harden the security of random number generation, we have also developed a library. It tries to improve random number generation by getting input from user. User touches a rectangular area on the device and touch's coordinates are used with platform's default random number generator to generate random number.

We run tests to select a library for our key derivation function, scrypt. For that two of the most popular scrpyt supporting Android libraries are tested. Various scrypt parameters are tested on actual devices to find its optimal parameters which provide good security without damaging usability.

We designed a complete solution for secure messaging. In addition to existing solutions, we added enterprise functionalities, hardened the security of random number generation, tested the performance of some widely used security libraries and their various parameters for the usability on mobile devices.

Chapter 5

Conclusion

In this thesis, we designed a secure mobile messaging solution. Some of the widely used solutions (Cryptocat, Telegram, Threema and Signal) are analyzed with respect to registration mechanisms, authentication methods, message encryption, perfect forward secrecy (PFS), disk encryption and reactions to possible man-in-the-middle (MitM) attacks. Their security protocols are explained, strengths and weaknesses are discussed.

We specified security requirements of a secure mobile messaging solution. When designing our solution, we referred to these requirements. Design rationales of each requirement and topic are discussed; test results are presented where applicable.

Our design supports end-to-end encrypted messaging with PFS support. PFS uses *advertise/acknowledge/send* message sequence for re-keying and there are no time or message limits; after a re-key is completed, new key's advertise message is send. All communications over the Internet use Transport Layer Security (TLS). Certificate pinning is also implemented against possible certificate authority compromises and MitM attacks. Frequently used AES-CBC and HMAC-SHA256 ciphers are chosen for message encryption and authentication. Two different

methods are selected for local disk encryption. Default one uses SQLCipher library which is easy to use and has good security design but greatly increases application package size (about 7MB). Alternative method is encrypting critical columns. This method is harder to implement and leaks database meta-data but application package is not bloated. scrypt is used as key derivation function. Its parameters are decided after testing execution times of various parameters on devices with different computation powers (Samsung Galaxy S II, LG Nexus 4, Samsung Galaxy Note II, LG Nexus 5, HTC Desire Eye and Samsung Galaxy S5). To improve the security of random number generation, we designed a library which collects user's touch coordinates and combine them with platform's default random number. This library is implemented as a precaution for possible vulnerabilities and to make random number guessing more difficult.

5.1 Future Work

We designed a complete secure messaging solution, but still some security and usability improvements can be made.

TouchRandom library can include timing data of users' touches to further increase the entropy of random number generation. Also, its "required number of coordinates" can be worked on to find an optimal value.

Instead of selecting scrpyt parameters to cover many low-end devices, dynamic parameters can be used. According to the device's computation power, parameters may be set so it would be harder to brute-force the keys which are created by high-end devices.

Client-to-client verification can be made more user-friendly. Instead of hexadecimal display of fingerprints; QR codes and colorful block visualizations (like Telegram's) can be displayed. Also, audio or video recordings can be sent as proofs (like Wickr's).

Bibliography

[1] "Off-the-record communication, or, why not to use pgp." https://otr.cypherpunks.ca/otr-codecon.pdf. Accessed: 2015-09-07.

[2] "key_image.jpg." https://telegram.org/img/key_image.jpg. Accessed: 2015-03-10.

[3] "Mtproto mobile protocol." https://core.telegram.org/mtproto. Accessed: 2015-03-10.

[4] "Threema cryptography whitepaper." https://threema.ch/press-files/cryptography_whitepaper.pdf. Accessed: 2015-03-05.

[5] "Advanced cryptographic ratcheting." https://whispersystems.org/blog/advanced-ratcheting/. Accessed: 2014-07-01.

[6] "scrypt: A new key derivation function." https://www.tarsnap.com/scrypt/scrypt-slides.pdf. Accessed: 2015-12-16.

[7] "Ott messaging traffic will be twice volume of p2p sms traffic this year." http://informa.com/media/press-releases-news/latest-news/ott-messaging-traffic-will-be-twice-volume-of-p2p-sms-traffic-this-year/. Accessed: 2015-12-04.

[8] "Edward snowden: Leaks that exposed us spy programme." http://www.bbc.com/news/world-us-canada-23123964. Accessed: 2015-12-04.

[9] "Nsa files decoded: Edward snowden's surveillance revelations explained." http://www.theguardian.com/world/interactive/2013/nov/01/

snowden-nsa-files-surveillance-revelations-decoded. Accessed: 2015-12-04.

[10] "Snowden leak: Nsa uses warrantless web surveillance to watch cyberattacks." https://www.rt.com/usa/265025-snowden-nsa-fbi-hackers/. Accessed: 2015-12-04.

[11] "Secret to prism program: Even bigger data seizure." http://bigstory.ap.org/article/secret-prism-success-even-bigger-data-seizure. Accessed: 2015-12-04.

[12] "Nsa prism program taps in to user data of apple, google and others." http://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data. Accessed: 2015-12-04.

[13] "U.s., british intelligence mining data from nine u.s. internet companies in broad secret program." https://www.washingtonpost.com/investigations/us-intelligence-mining-data-from-nine-us-internet-companies-in-broad-secret 2013/06/06/3a0c0da8-cebf-11e2-8845-d970ccb04497_story.html. Accessed: 2015-12-04.

[14] "Diginotar reports security incident." https://www.vasco.com/company/about_vasco/press_room/news_archive/2011/news_diginotar_reports_security_incident.aspx. Accessed: 2015-04-26.

[15] "Comodo report of incident - comodo detected and thwarted an intrusion on 26-mar-2011." https://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html. Accessed: 2015-04-26.

[16] "Google discovers fraudulent digital certificate issued for its domain." http://www.wired.com/2013/01/google-fraudulent-certificate/. Accessed: 2015-04-26.

[17] H. Krawczyk, R. Canetti, and M. Bellare, "Hmac: Keyed-hashing for message authentication," 1997.

[18] W. Diffie and M. E. Hellman, "New directions in cryptography," *Information Theory, IEEE Transactions on*, vol. 22, no. 6, pp. 644–654, 1976.

[19] A. W. H. E. Nils Gura, Arun Patel and S. C. Shantz, "Comparing elliptic curve cryptography and rsa on 8-bit cpus.," in *6th International Workshop on Cryptographic Hardware and Embedded Systems*, (Boston, Massachusetts), Aug. 2004.

[20] "Heartbleed bug." `http://heartbleed.com/`. Accessed: 2015-10-18.

[21] B. Kaliski, "PKCS #5: Password-Based Cryptography Specification Version 2.0." RFC 2898 (Informational), Sept. 2000.

[22] N. Provos and D. Mazières, "A future-adaptive password scheme," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, (Berkeley, CA, USA), pp. 32–32, USENIX Association, 1999.

[23] C. Percival, "Stronger key derivation via sequential memory-hard functions," *Self-published*, pp. 1–16, 2009.

[24] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile." RFC 5280 (Proposed Standard), May 2008. Updated by RFC 6818.

[25] S. Chokhani, W. Ford, R. Sabett, C. Merrill, and S. Wu, "Internet X.509 Public Key Infrastructure Certificate Policy and Certification Practices Framework." RFC 3647 (Informational), Nov. 2003.

[26] "Let's encrypt - free ssl/tls certificates." `https://letsencrypt.org/`. Accessed: 2015-12-28.

[27] "Let's encrypt is trusted - let's encrypt - free ssl/tls certificates." `https://letsencrypt.org/2015/10/19/lets-encrypt-is-trusted.html`. Accessed: 2015-12-28.

[28] "Firesheep." `https://codebutler.github.io/firesheep/`. Accessed: 2015-12-27.

[29] N. Borisov, I. Goldberg, and E. Brewer, "Off-the-record communication, or, why not to use pgp," in *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, WPES '04, (New York, NY, USA), pp. 77–84, ACM, 2004.

[30] "Cryptocat." `https://crypto.cat`. Accessed: 2015-12-27.

[31] "Chatsecure encrypted messenger for ios and android." `https://chatsecure.org/`. Accessed: 2015-12-22.

[32] "jitsi.org — jitsi." `https://jitsi.org`. Accessed: 2015-12-27.

[33] "Axolotl ratchet." `https://github.com/trevp/axolotl/wiki`. Accessed: 2014-06-29.

[34] "Avast blog tens of thousands of americans sell themselves online every day." `http://blog.avast.com/2014/07/08/tens-of-thousands-of-americans-sell-themselves-online-every-day/`. Accessed: 2015-04-25.

[35] "Avast." `https://www.avast.com`. Accessed: 2015-04-24.

[36] M. Y. C. Wei, L. M. Grupp, F. E. Spada, and S. Swanson, "Reliably erasing data from flash-based solid state drives.," in *FAST*, vol. 11, pp. 8–8, 2011.

[37] "Multiparty protocol specification." `https://github.com/cryptocat/cryptocat/wiki/Multiparty-Protocol-Specification`. Accessed: 2015-04-10.

[38] "Add certificate pinning for crypto.cat. (closed)." `https://codereview.chromium.org/11358199/`. Accessed: 2015-04-10.

[39] "Long-term keys needed across platforms (including browsers)." `https://github.com/cryptocat/cryptocat/issues/580`. Accessed: 2015-04-10.

[40] "Github." `https://github.com/`. Accessed: 2015-04-10.

[41] "Off-the record messaging protocol in javascript." `https://github.com/arlolra/otr`. Accessed: 2015-04-10.

[42] "Off-the-record messaging protocol version 3 - draft." https://otr.cypherpunks.ca/Protocol-v3-4.0.0.html. Accessed: 2015-04-14.

[43] "mpotr project plan." https://github.com/cryptocat/cryptocat/wiki/mpOTR-Project-Plan. Accessed: 2015-04-10.

[44] "Meet telegram, a secure messaging app from the founders of vk, russia?s largest social network." http://techcrunch.com/2013/10/27/meet-telegram-a-secure-messaging-app-from-the-founders-of-vk-russias-larges Accessed: 2015-03-10.

[45] "Using telegram api." https://core.telegram.org/api. Accessed: 2015-03-10.

[46] "Telegram faq." https://telegram.org/faq. Accessed: 2015-03-10.

[47] "Telegram - android apps on google play." https://play.google.com/store/apps/details?id=org.telegram.messenger. Accessed: 2015-03-10.

[48] "Local encryption?." https://github.com/DrKLO/Telegram/issues/889. Accessed: 2015-03-10.

[49] "$200,000 to the first person to break telegram — hacker news." https://news.ycombinator.com/item?id=6931457. Accessed: 2015-03-10.

[50] "Moxie marlinspike ¿¿ blog ¿¿ a crypto challenge for the telegram developers." http://www.thoughtcrime.org/blog/telegram-crypto-challenge/. Accessed: 2015-03-10.

[51] "Crypto fails ? telegram's cryptanalysis contest." http://www.cryptofails.com/post/70546720222/telegrams-cryptanalysis-contest. Accessed: 2015-03-10.

[52] "Crowdsourcing a more secure future." https://telegram.org/blog/crowdsourcing-a-more-secure-future. Accessed: 2015-03-10.

[53] "Some securerandom thoughts — android developers blog." http://android-developers.blogspot.ru/2013/08/some-securerandom-thoughts.html. Accessed: 2015-03-10.

[54] "$300,000 for cracking telegram encryption." https://telegram.org/blog/cryptocontest. Accessed: 2015-03-10.

[55] "Crypto contest ends." https://telegram.org/blog/cryptocontest-ends. Accessed: 2015-03-10.

[56] J. Jakobsen and C. Orlandi, "On the cca (in) security of mtproto,"

[57] J. Katz and Y. Lindell, *Introduction to modern cryptography*. CRC Press, 2014.

[58] "Stalking anyone on telegram." https://oflisback.github.io/telegram-stalking/. Accessed: 2015-12-20.

[59] "Threema on the app store on itunes." https://itunes.apple.com/us/app/threema/id578665578. Accessed: 2014-07-15.

[60] "Threema - android apps on google play." https://play.google.com/store/apps/details?id=ch.threema.app. Accessed: 2015-03-07.

[61] "Threema shop (android)." https://shop.threema.ch/. Accessed: 2014-07-15.

[62] "Threema - windows apps on microsoft store." https://www.microsoft.com/en-us/store/apps/threema/9nblggh095s5. Accessed: 2015-12-20.

[63] "Bye bye, whatsapp: Germans switch to threema for privacy reasons." http://techcrunch.com/2014/02/21/bye-bye-whatsapp-germans-switch-to-threema-for-privacy-reasons/. Accessed: 2014-07-15.

[64] "Does threema provide forward secrecy?." https://threema.ch/en/faq.html. Accessed: 2014-07-15.

[65] "Threema - seriously secure messaging - is threema secure against man-in-the-middle (mitm) attacks?." https://threema.ch/en/faq/crypto_mitm. Accessed: 2015-03-05.

[66] "Threema security assessment." https://www.os3.nl/_media/2013-2014/courses/ssn/projects/threema_report.pdf. Accessed: 2015-12-18.

[67] "Nacl: Networking and cryptography library." http://nacl.cr.yp.to/. Accessed: 2015-03-07.

[68] "Threema encryption validation." https://threema.ch/validation/. Accessed: 2015-03-05.

[69] "Microsoft onedrive access your files from anywhere." https://onedrive.live.com/about/en-en/. Accessed: 2015-12-20.

[70] "Textsecure private messenger." https://play.google.com/store/apps/details?id=org.thoughtcrime.securesms. Accessed: 2014-07-08.

[71] "Cyanogenmod — android community operating system." http://www.cyanogenmod.org/. Accessed: 2014-07-08.

[72] "Textsecure, now with 10 million more users." https://whispersystems.org/blog/cyanogen-integration/. Accessed: 2014-07-08.

[73] "Signal - private messenger on the app store on itunes." https://itunes.apple.com/us/app/signal-private-messenger/id874139669. Accessed: 2015-03-03.

[74] T. Berners-Lee, R. Fielding, and H. Frystyk, "Hypertext Transfer Protocol – HTTP/1.0." RFC 1945 (Informational), May 1996.

[75] "Google cloud messaging for android." https://developer.android.com/google/gcm/index.html. Accessed: 2014-06-29.

[76] "bouncycastle.org." https://www.bouncycastle.org/. Accessed: 2015-12-16.

[77] "Api protocol." https://github.com/WhisperSystems/TextSecure-Server/wiki/API-Protocol. Accessed: 2014-06-29.

[78] P. R. Zimmermann, *The Official PGP User's Guide*. Cambridge, MA, USA: MIT Press, 1995.

[79] "Forward secrecy for asynchronous messages." https://whispersystems.org/blog/asynchronous-security/. Accessed: 2014-06-29.

[80] "Protocolv2." https://github.com/WhisperSystems/TextSecure/wiki/ProtocolV2. Accessed: 2014-06-30.

[81] H. Krawczyk and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)." RFC 5869 (Informational), May 2010.

[82] "Bring back the encrypted backup! issue #1705 whispersystems/signal-android." https://github.com/WhisperSystems/Signal-Android/issues/1705. Accessed: 2015-12-20.

[83] "Saying goodbye to encrypted sms/mms." https://whispersystems.org/blog/goodbye-encrypted-sms/. Accessed: 2015-04-15.

[84] "Smssecure." https://github.com/SMSSecure/SMSSecure. Accessed: 2015-04-15.

[85] "Contributions - jul 3, 2011 dec 22, 2015." https://github.com/cryptocat/cryptocat/graphs/contributors. Accessed: 2015-12-22.

[86] "Whatsapp just switched on end-to-end encryption for hundreds of millions of users." http://www.wired.com/2014/11/whatsapp-encrypted-messaging. Accessed: 2015-12-22.

[87] "Wickr — the most trusted messenger in the world." https://www.wickr.com/. Accessed: 2015-12-22.

[88] "surespot — encrypted chat messenger — 100https://www.surespot.me/. Accessed: 2015-12-22.

[89] "surespot encrypted messenger - android apps on google play." https://play.google.com/store/apps/details?id=com.twofours.surespot. Accessed: 2015-12-22.

[90] "Wickr-top secret messenger - android apps on google play." https://play.google.com/store/apps/details?id=com.mywickr.wickr2. Accessed: 2015-12-22.

[91] "Chatsecure - android apps on google play." https://play.google.com/store/apps/details?id=info.guardianproject.otr.app.im. Accessed: 2015-12-22.

[92] "Secure messaging scorecard — electronic frontier foundation." https://www.eff.org/secure-messaging-scorecard. Accessed: 2015-12-23.

[93] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: Validating ssl certificates in non-browser software," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, (New York, NY, USA), pp. 38–49, ACM, 2012.

[94] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)." RFC 7519 (Proposed Standard), May 2015.

[95] "Fips pub 197, advanced encryption standard (aes)," 2001. U.S.Department of Commerce/National Institute of Standards and Technology.

[96] A. Biryukov, O. Dunkelman, N. Keller, D. Khovratovich, and A. Shamir, "Key recovery attacks of practical complexity on aes variants with up to 10 rounds." Cryptology ePrint Archive, Report 2009/374, 2009. http://eprint.iacr.org/.

[97] N. S. Thakur, "Forensic analysis of whatsapp on android smartphones," 2013.

[98] "¡manifest¿ — android developers." https://developer.android.com/guide/topics/manifest/manifest-element.html#uid. Accessed: 2015-12-12.

[99] "Sqlcipher - zetetic." https://www.zetetic.net/sqlcipher/sqlcipher-for-android/. Accessed: 2015-12-12.

[100] "Proguard — android developers." https://developer.android.com/tools/help/proguard.html. Accessed: 2015-12-12.

[101] "Spongy castle by rtyley." https://rtyley.github.io/spongycastle/. Accessed: 2015-12-16.

[102] "wg/scrypt github." https://github.com/wg/scrypt. Accessed: 2015-12-16.

[103] "facebook/device-year-class github." https://github.com/facebook/device-year-class. Accessed: 2015-12-16.