Aarhus University

Computer Science

Master's thesis

A practical cryptanalysis of the Telegram messaging protocol

Author:
Jakob Bjerre Jakobsen
(20102095)

Supervisor:
Claudio Orlandi

September 2015

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

The number one rule for cryptography is never create your own crypto. Instant messaging application Telegram has disregarded this rule and decided to create an original message encryption protocol. In this work we have done a thorough cryptanalysis of the encryption protocol and its implementation. We look at the underlying cryptographic primitives and how they are combined to construct the protocol, and what vulnerabilities this has. We have found that Telegram does not check integrity of the padding applied prior to encryption, which lead us to come up with two novel attacks on Telegram. The first of these exploits the unchecked length of the padding, and the second exploits the unchecked padding contents. Both of these attacks break the basic notions of IND-CCA and INT-CTXT security, and are confirmed to work in practice. Lastly, a brief analysis of the similar application TextSecure is done, showing that by using well known primitives and a proper construction provable security is obtained. We conclude that Telegram should have opted for a more standard approach.

Acknowledgements

First of all I would like to thank my former office partner Morten Djernæs Bech who had the energy to have helpful discussions with me about my thesis, even though the deadline for his own thesis was much closer than mine. I would like to thank Frederik Bitsch Kirk for proofreading and spell checking my entire work, truly his English skill surpasses mine. And finally I would like give great thanks to my supervisor Claudio Orlandi for helping me realize my idea for this project, and for his guidance and lengthy discussions with me that led to interesting discoveries.

Contents

Chapter 1

Introduction

Telegram is an instant messaging service designed for mobile devices, that as of May 2015 had 62 million monthly active users [23] and is used to deliver ten milliard[1] messages daily [29]. It offers two conversation modes, the first being the regular chat mode in which all messages can be read by the server and will be stored, allowing for synchronization between devices and group chats. The second mode, called secret chat, is an end-to-end encrypted chat for only two parties. Messages are sent through the server, but cannot be read by it and are claimed to not be stored at any time.
Instead of using proven cryptographic constructs, Telegram opted to create its own original protocol known as MTProto. This is built on weaker primitives, but in such a way that known attacks do not apply.
The official Telegram client application is open source, allowing for full auditing, but the server software is not. According to the Electronic Frontier Foundation [9], Telegram was audited in February 2015 and the secret chat mode achieved full marks for its security. This work will go in-depth with the secret chat mode, analyzing potential weaknesses and propose attacks to exploit these as well as proposing improvements.

Goals The goal of this work is to give a thorough analysis of the security of the Telegram instant messaging application and its message encryption protocol MTProto. We will look at the cryptographic primitives used, and how the message encryption protocol is constructed from these. We will see which notions of security this protocol provides, and which it does not. With this knowledge we will propose attacks on the Telegram messaging protocol, exploiting the found weaknesses. Finally, we will propose improvements to the protocol that solve found issues and mitigate proposed attacks.

[1] Or ten US billion.

1.1 Chapter overview

1.1.1 Chapter 2: Preliminaries

In the preliminaries chapter we will give an overview of the cryptographic primitives that Telegram is built upon. First we will introduce symmetric encryption cryptosystems and more specifically Advanced Encryption Standard and Infinite Garble Extension, the mode of operation employed by Telegram. We will define relevant security notions starting from Indistinguishability of encryptions under Chosen-Plaintext Attack and leading up to Authenticated Encryption. On the way we will cover Integrity of Plaintexts and Ciphertexts, unforgeable Message Authentication Codes, and we will also define Blockwise-Adaptive Indistinguishability of encryptions.
We will also give an overview of hash functions and define collision resistance, and relate this to pre-image and 2nd pre-image resistance as well as identical-prefix collision resistance. Finally, we will briefly introduce Secure Hash Algorithm SHA1, and explain what it means for a hash function to be cryptographically broken.

1.1.2 Chapter 3: Protocols

In the protocols chapter we will walk through the flow of the Telegram application. We will first cover how a new user registers a device to the Telegram server, and how the two exchange an authorization key. Next we will explain how two users exchange keys and initiate an end-to-end encrypted secret chat. We will then explain how symmetric encryption keys are derived, and how clients encrypt and decrypt messages using Telegram's own MTProto scheme. Furthermore, we will cover how the message counter in Telegram is constructed, and how Forward Secrecy is provided. Finally, we will see how MTProto provides backwards compatibility in order for older clients to remain compatible with clients on newer versions.
Throughout this chapter we will also outline some inconsistencies between how Telegram explains their encryption scheme on their website and how it is actually implemented, which will lead to some interesting exploits in the next chapter.

1.1.3 Chapter 4: Results of analysis

In the results chapter we will outline vulnerabilities in MTProto, and construct attacks that exploit these in order to break security properties of MTProto. We will show how the padding scheme employed leads to two attacks

that break both Integrity of Ciphertexts and Indistinguishability of encryptions under Chosen-Ciphertext Attack. The first attack exploits the fact that padding length is never checked, the second exploiting the fact that the padding has no integrity.
We will also describe how a malicious server could take advantage of older, still supported versions of MTProto to freely inject messages into end-to-end encrypted conversations.
Finally, we will look at how MTProto could be vulnerable to timing attacks because of how message content is checked upon decryption, but such an attack has not been implemented for validation.

1.1.4 Chapter 5: Experimental validation

In the experimental validation chapter we will implement and verify the effectiveness of the attacks as described in the results chapter. We will briefly explain the test setup, and for each attack we will show how it is implemented, how the test was run, and what the outcome was.
We confirm that the padding extension attack has probability 1 of succeeding in breaking IND-CCA security, whereas the padding plaintext collision attack succeeds with non-negligible probability 2^{-32}.
We also confirm that by using an older version of MTProto that relies on trusting the server for message counters of end-to-end encrypted chats, a malicious server can freely inject messages.
Finally, for each of the attacks we will show how to patch the vulnerability and mitigate the attack, and see how these patches will affect client compatibility.

1.1.5 Chapter 6: Known attacks

In the known attacks chapter we will cover attacks that are known to affect the cryptographic primitives of MTProto, and see if these also apply to MTProto itself. Specifically we look at blockwise-adaptive IND-CPA for Infinite Garble Extension and the cost of finding collisions for SHA1, and how this affects the security of MTProto.
We will also describe attacks constructed specifically for MTProto, including a former malicious server attack that would let the server learn two users' shared secret, even though this has been patched out. The other two attacks are third party man-in-the-middle attacks, both of which will grant the attacker full access to all messages. We will see if these attacks are feasible to carry out.

1.1.6 Chapter 7: Proven crypto alternative

In the proven crypto alternative chapter we will look at another instant messaging application that does have proven Authenticated Encryption. Specifically we will look at how TextSecure operates, which primitives it uses and how these are combined to guarantee confidentiality, integrity and authenticity. We will relate this to Telegram and see what Telegram can take from this standard approach and use to improve upon its own homegrown approach.

1.1.7 Contributions

The contributions of this work are two novel vulnerabilities that we found in the protocol MTProto, proving it insecure under basic notions.

These findings were communicated to the Telegram team on the 3rd of September 2015, and we have yet to get a response.

Chapter 2

Preliminaries

2.1 Notation

Throughout this work we will use consistent notation. We write $\leftarrow$ for assignment. '=' is used for conditional checks. Variable names are written in *italic*.

Encryption under key K is written as Enc_K and similarly decryption is Dec_K.

We will refer to full plaintext messages with an uppercase M, or M_b for a bit $b \in \{0, 1\}$ in cases where one of two messages is chosen. Similarly, a full ciphertext is referred to as C. $|M|$ refers to the length of message M.

For individual blocks of a message, m_{bi} is the i'th block of message M_b, and c_i is the ciphertext of the i'th block.

The string concatenation operator is denoted as $||$, and XOR is $\oplus$.

2.2 Symmetric-key cryptosystems

Symmetric- or private-key cryptosystems are designed to provide confidentiality, keeping the contents of a data block secret. These are generally computationally secure, meaning that given enough time and computing power an adversary could recover the plaintext, but in practice this task is nearly impossible to carry out. These systems are defined as follows, from Katz et al. [13]:

2.2.1 Def. private-key encryption scheme

A private-key encryption scheme is a tuple of probabilistic polynomial-time algorithms (Gen, Enc, Dec) such that:

1. The key-generation algorithm Gen takes as input the security parameter 1^n and outputs a key k; we write this as $k \leftarrow \text{Gen}(1^n)$ (thus emphasizing the fact that Gen is a randomized algorithm). We will assume without loss of generality that any key k output by $\text{Gen}(1^n)$ satisfies $|k| \geq n$.

2. The encryption algorithm Enc takes as input a key k and a plaintext message $m \in \{0,1\}^*$ and outputs a ciphertext c. Since Enc may be randomized, we write this as $c \leftarrow \text{Enc}_k(m)$.

3. The decryption algorithm Dec takes as input key k and a ciphertext c and outputs a message m. We assume that Dec is deterministic, and so write this as $m \leftarrow \text{Dec}_k(c)$.

It is required that for every n, every k output by $\text{Gen}(1^n)$, and every $m \in \{0,1\}^*$, it holds that $\text{Dec}_k(\text{Enc}_k(m)) = m$.

2.2.2 Advanced Encryption Standard (AES)

AES is the successor of the Data Encryption Standard (DES) from 1977 which was eventually too easy to break with a bruteforce attack because of its 56-bit keys. It has been the standard since its adoption in 2001, and to this date no computationally feasible key recovery attack has been published.

AES is a symmetric-key block cipher with block size 128 bit and keys ranging from 128 to 256 bits. The internal components of AES are beyond the scope of this work, and we will treat it as a black box for encryption.

2.2.3 Infinite Garble Extension (IGE)

Infinite Garble Extension is a mode of operation for block ciphers in symmetric-key encryption. IGE blocks are chained as follows:

$$c_{i-1} \leftarrow \text{f}_K(m_i \oplus c_{i-1}) \oplus m_{i-1}$$

f_K being the block cipher encryption function using key K.

Typically, a mode of operation for block ciphers needs a block length initialization vector, which we will later see is the case for CBC and CTR

mode. But as we see in Savard [18], IGE needs both an initial plaintext block m_0 and ciphertext block c_0. The original specification says the initial ciphertext block is derived using a second random key K_0 : $c_0 = \mathrm{f}_{K_0}(m_0)$, but the OpenSSL implementation allows for it to be an arbitrary block given as a parameter.

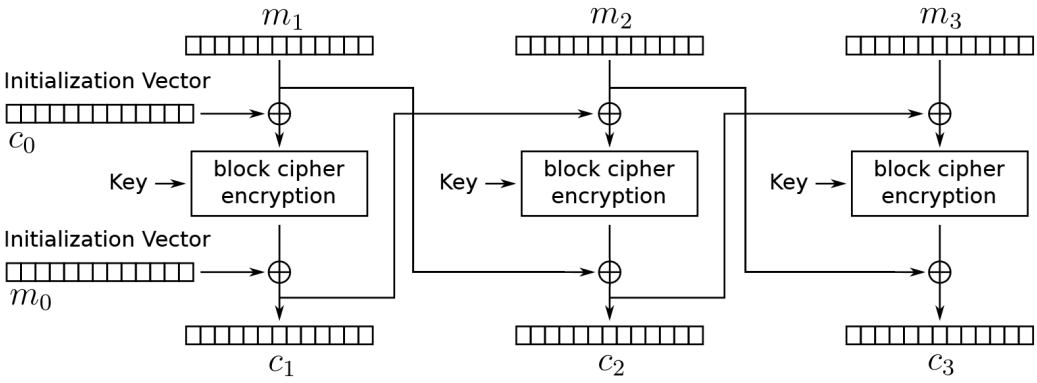

Figure 2.1: Diagram of IGE mode of operation for encryption.

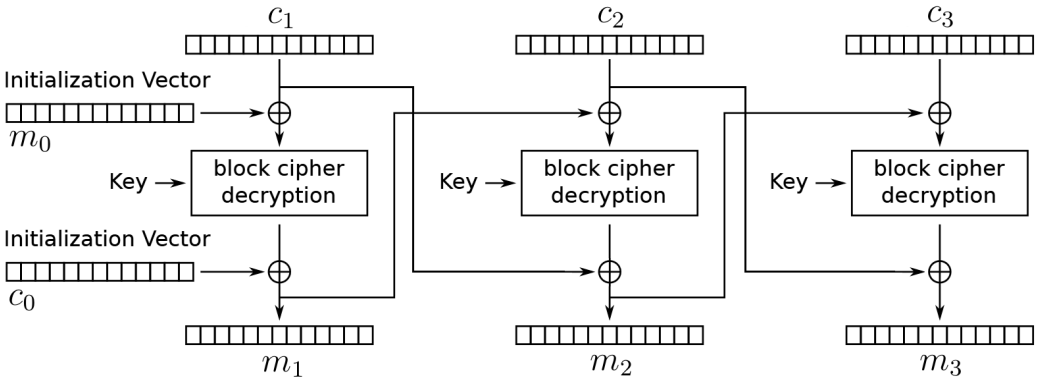

Figure 2.2: Diagram of IGE mode of operation for decryption.

Writing out the operations for encrypting each block we get the following, as visualized in figure 2.1:

$$c_1 \leftarrow m_0 \oplus \mathrm{Enc}_K(m_1 \oplus c_0)$$
$$c_2 \leftarrow m_1 \oplus \mathrm{Enc}_K(m_2 \oplus m_0 \oplus \mathrm{Enc}_K(m_1 \oplus c_0))$$
$$c_3 \leftarrow m_2 \oplus \mathrm{Enc}_K(m_3 \oplus m_1 \oplus \mathrm{Enc}_K(m_2 \oplus m_0 \oplus \mathrm{Enc}_K(m_1 \oplus c_0)))$$
$$c_i \leftarrow m_{i-1} \oplus \mathrm{Enc}_K(m_i \oplus m_{i-2} \oplus \mathrm{Enc}_K(m_{i-1} \oplus \ldots m_0 \oplus \mathrm{Enc}_K(m_1 \oplus c_0) \ldots))$$

Writing out the operations for decrypting each block we get the following, as visualized in figure 2.2:

$$
\begin{aligned}
m_1 &\leftarrow c_0 \oplus \mathrm{Dec}_K(c_1 \oplus m_0) \\
m_2 &\leftarrow c_1 \oplus \mathrm{Dec}_K(c_2 \oplus c_0 \oplus \mathrm{Dec}_K(c_1 \oplus m_0)) \\
m_3 &\leftarrow c_2 \oplus \mathrm{Dec}_K(c_3 \oplus c_1 \oplus \mathrm{Dec}_K(c_2 \oplus c_0 \oplus \mathrm{Dec}_K(c_1 \oplus m_0))) \\
m_i &\leftarrow c_{i-1} \oplus \mathrm{Dec}_K(c_i \oplus c_{i-2} \oplus \mathrm{Dec}_K(c_{i-1} \oplus \ldots c_0 \oplus \mathrm{Dec}_K(c_1 \oplus m_0) \ldots))
\end{aligned}
$$

An article on the implementation of IGE in OpenSSL [14] states that the mode of operation IGE «has the property that errors are propagated forward indefinitely».

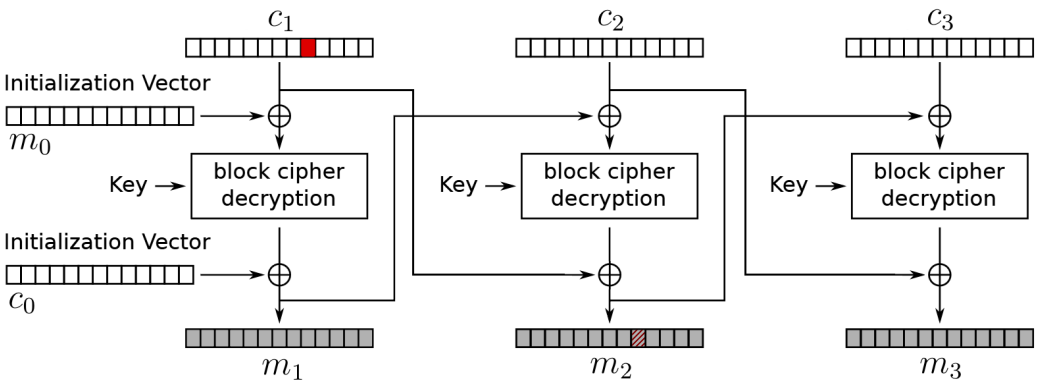

Figure 2.3: Non-malleability of IGE causing every block after the modification to become garbled.

Figure 2.3 shows that flipping one bit in one ciphertext block would cause the decrypted block to garble as well as all the following blocks. It is also possible to use bi-directional IGE, garbling the entire plaintext output. This can be used to provide plaintext integrity, guaranteeing the contents of a message has not been tampered with, by appending a block of all 0's at the end and checking this upon decryption.

Cipher Block Chaining (CBC) and malleability A more common mode of operation like IGE that does not have this garbling property, and as shown in figure 2.4, flipping one bit in a ciphertext block will only cause the corresponding plaintext block to garble as well as flipping the bit in the same position of the following plaintext block. This makes CBC malleable,

an often undesirable property as it can be exploited to break plaintext integrity.

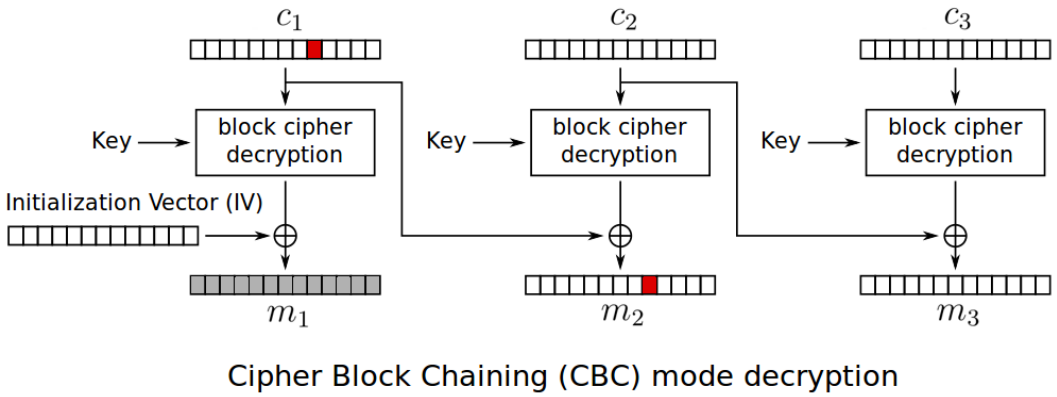

Figure 2.4: Malleability of CBC mode allow modifying one plaintext output block, at the cost of garbling the one before it.

This is bad if evil Eve knows Alice is sending Bob the message "Hi Bob, please transfer $100 to Eve", Eve could capture the encrypted ciphertext and modify the block before the one containing the value, so that the decrypted message becomes "[garbled] transfer $999 to Eve".

We have seen malleability exploited in the BEAST attack on SSL/TLS, as described by Paterson [15]. Here we saw that the padding of length n bytes consisted of n copies of the byte $n-1$, and this padding structure was checked before the message integrity was checked from the MAC. This little difference in timing and the malleability of CBC mode lead to a Padding Oracle Attack, allowing an adversary to recover the full plaintext one byte at a time.

2.2.4 Counter mode (CTR)

The counter mode of operation is different from IGE and CBC that we saw above in that it does not do any block chaining. What it does instead is to extend the key into an arbitrarily long key stream and XOR this with the plaintext. Generating this key stream is done by first choosing a random nonce of one block length. This nonce is encrypted under the encryption key, which gives the first block of the key stream. The stream is then XOR'd with the plaintext, outputting one ciphertext block. For every following plaintext block, the counter is incremented corresponding to the position of the block in the plaintext message, and the same process is used.
The decryption scenario is identical except the plaintext is switched with

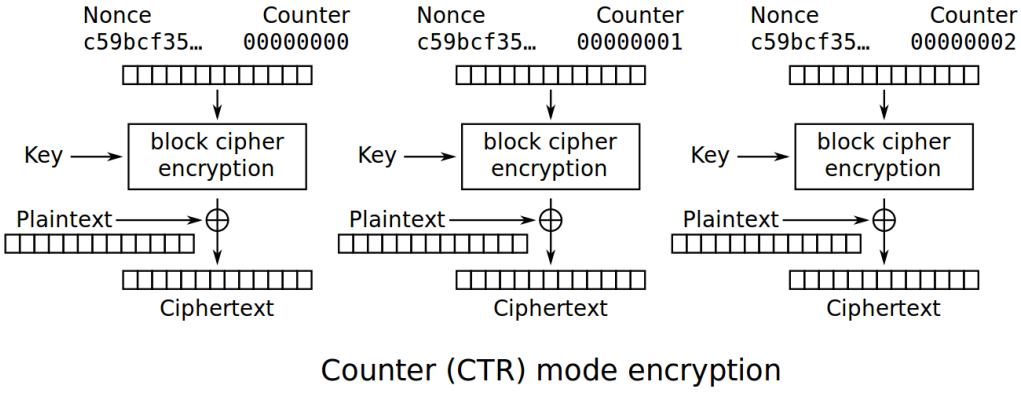

Figure 2.5: Encryption using counter mode.

the ciphertext. It's easy to see that like CBC, CTR mode is malleable but with no blocks being garbled in the process. The lack of block chaining also makes counter mode trivially parallelizable, leading to higher performance on multi-core systems.
Rather than adding padding to the plaintext, the key stream generated with CTR mode can simply be truncated to match the plaintext length.

2.3 Security definitions

In this section we will define the notions of security that a cryptosystem should ideally provide, and hold it against MTProto in later chapters.

2.3.1 Introduction to oracles

To help explain the security definitions we will use oracle games. These are hypothetical scenarios where a probabilistic polynomial-time adversary $\mathcal{A}$ is challenged by an oracle $\mathcal{O}$ to output an answer that meets certain criteria. Generally speaking, $\mathcal{A}$ must output a satisfying answer with non-negligible probability for a security parameter n.

Left-Or-Right oracles In the case of LOR-oracles, $\mathcal{O}$ chooses a bit b at random, and $\mathcal{A}$ has to be able to output if it is 1 or 0 with probability of winning non-negligibly higher than that of guessing at random, based on security parameter n.

2.3.2 Definition: IND-CPA security

Indistinguishability of encryptions under chosen-plaintext attack is a basic notion of security for cryptosystems. It describes the inability of distinguishing encrypted plaintext, thereby providing confidentiality, which in some cases will be a sufficient security notion. We use a LOR-oracle game for this definition, and the idea is that $\mathcal{A}$ has to be able to distinguish which plaintext of his choice that $\mathcal{O}$ has encrypted. From Katz et al. [13]:
For a symmetric key encryption system $\Pi \leftarrow$(Gen, Enc, Dec) with security parameter n:

1. A key K is generated by running $\text{Gen}(1^n)$.
2. The adversary $\mathcal{A}$ is given input 1^n and oracle access to $\text{Enc}_K(\cdot)$, and outputs a pair of messages M_0, M_1 of the same length l.
3. A random bit $b \leftarrow \{0,1\}$ is chosen, and then a ciphertext $C \leftarrow \text{Enc}_K(M_b)$ is computed and given to $\mathcal{A}$. We call C the challenge ciphertext.
4. The adversary $\mathcal{A}$ continues to have oracle access to $\text{Enc}_K(\cdot)$, and outputs a bit b'.
5. The output of the experiment is defined to be 1 if $b' = b$, and 0 otherwise.

We say the symmetric encryption scheme Π has indistinguishable encryptions under chosen-plaintext attack if for all probabilistic polynomial time adversaries $\mathcal{A}$ there exists a negligible function $\text{negl}(\cdot)$ such that

$$Pr[\text{Experiment outputs 1}] \leq \frac{1}{2} + \text{negl}(n)$$

eg. probability of winning is only negligibly better than guessing at random.

This definition assumes the cryptosystem is probabilistic, as it would be trivial to win in a deterministic setting by sending M_0 to the oracle a second time and seeing if the returned $C_i = C$ which implies $b = 0$, otherwise $b = 1$.

IND-CPA is a strong notion in that it makes no assumption about the plaintext, unlike weaker notions such as Known-Plaintext Attack in which the adversary does not get to choose the messages for the challenge ciphertexts, but it can break under certain circumstances. Take for instance an adversary that is given access to each individual ciphertext block as they are computed,

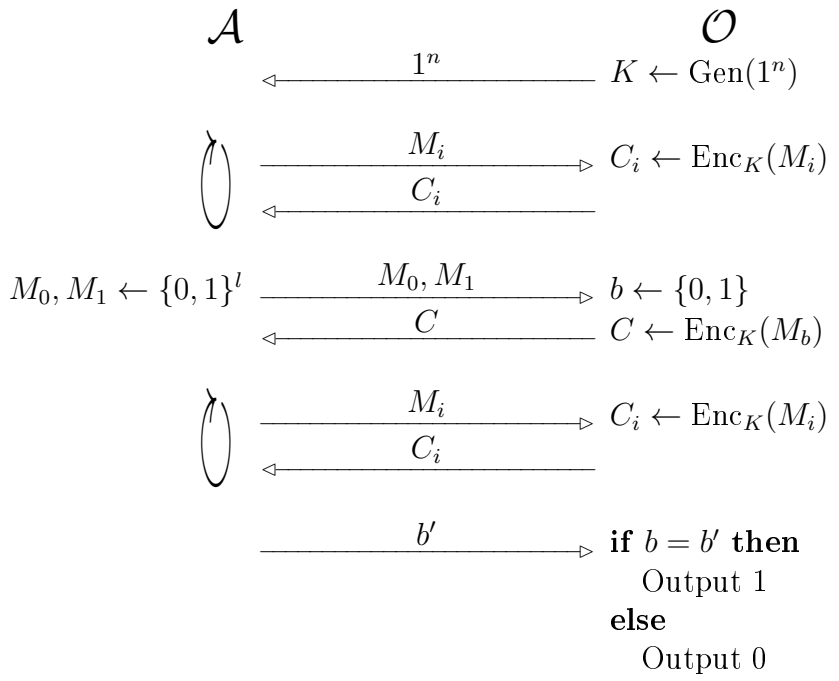

Figure 2.6: LOR-oracle IND-CPA game.

before inputting the next plaintext block. We will see that this can be broken even if the underlying cryptosystem has IND-CPA, which is why we will define blockwise-adaptive IND-CPA, or IND-BACPA.

2.3.3 Definition: general IND-BACPA

Indistinguishability of encryptions under blockwise-adaptive chosen-plaintext attack. From Bard [4]:

«The name "blockwise-adaptive" alludes to the capability of the attacker to view the results of inserting one or more blocks of choice into a plaintext message before deciding on the next block, thus permitting the attacker to "adapt" the attack based upon those observations.»

It may not be intuitive that this scenario can exist in practice, but it is the case for some online services where the server will encrypt each block as soon as it's uploaded, such as SSH. The definition is described by the following LOR-oracle game:
For a symmetric key encryption system $\Pi \leftarrow$(Gen, Enc, Dec) with security parameter n:

1. A key K is generated by running $\text{Gen}(1^n)$. A random bit $b \leftarrow \{0,1\}$ is chosen.
2. The adversary $\mathcal{A}$ is given input 1^n and oracle access to $\text{Enc}_K(\cdot)$, and sends a pair of message blocks m_{01}, m_{11} of fixed length.
3. A ciphertext block $c_1 \leftarrow \text{Enc}_K(m_{b1})$ is computed and given to $\mathcal{A}$.
4. After observing c_1, $\mathcal{A}$ now sends the next pair of plaintext blocks m_{02}, m_{12} and receives c_2. After sending polynomially many block pairs, $\mathcal{A}$ outputs a bit b'.
5. The output of the experiment is defined to be 1 if $b' = b$, and 0 otherwise.

We say the symmetric encryption scheme Π has indistinguishable encryptions under blockwise-adaptive chosen-plaintext attack if for all probabilistic polynomial time adversaries $\mathcal{A}$ there exists a negligible function $\text{negl}(\cdot)$ such that

$$Pr[\text{Experiment outputs 1}] \leq \frac{1}{2} + \text{negl}(n)$$

This is the notion of general IND-BACPA security, but another variant of it exists called primitive IND-BACPA. The only difference here is that the

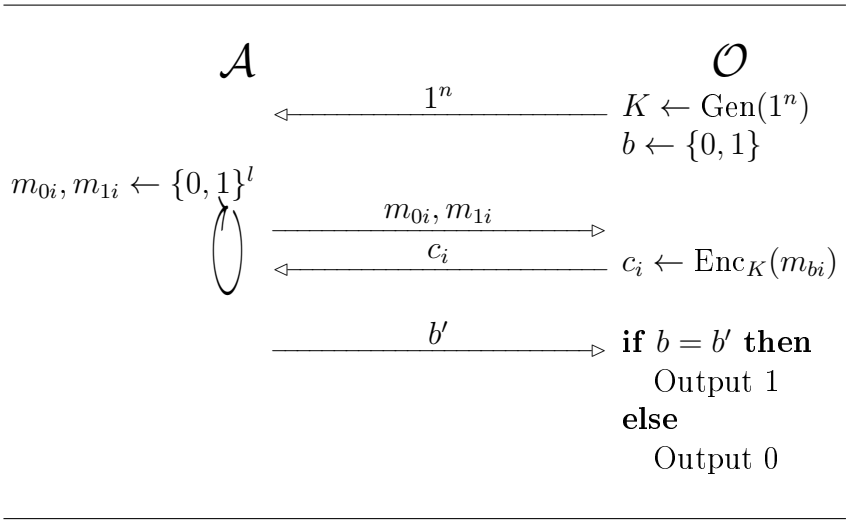

Figure 2.7: LOR-oracle IND-BACPA game.

adversary is only allowed to output plaintext blocks $m_{0i} \neq m_{1i}$ once, and for any other query it must hold that $m_{0i} = m_{1i}$. We will use this definition in section 6.1.1 where we will show that IGE does not have indistinguishable encryptions under the primitive BACPA notion.

This gives us a strong definition for encryption oracles, but we have not considered that the adversary might also have access to a decryption oracle.

Take for example the encryption function $\text{Enc}_K(M) = \text{F}_K(r) \oplus M$ where $|M| = n$, r is a random nonce and F_K is a pseudo-random function. An adversary now sends $M_0 \leftarrow 0^n$, $M_1 \leftarrow 1^n$. The oracle will now choose $b \leftarrow \{0,1\}$ and return $C \leftarrow \text{F}_K(r) \oplus M_b$. This system is CPA secure but not CCA secure. The oracle will not decrypt the challenge ciphertext C, but if the adversary was to flip the first bit of C, the oracle will gladly decrypt it. Now the returned plaintext message will be either 10^{n-1} or 01^{n-1}, making it easy to see if b is 1 or 0.

This is why it is generally not considered sufficient for a cryptosystem to have IND-CPA security, and stronger notions are required. Which leads us to chosen-ciphertext security.

2.3.4 Definition: IND-CCA

Indistinguishability of encryptions under chosen-ciphertext attack takes into account that the adversary could have access to decryption as well as encryption. This results in a stronger notion than IND-CPA, and it is described as a LOR-oracle game by Katz et al. [13]:
For a symmetric key encryption system $\Pi \leftarrow$(Gen, Enc, Dec) with security parameter n:

1. A key K is generated by running $\text{Gen}(1^n)$.
2. The adversary $\mathcal{A}$ is given input 1^n and oracle access to $\text{Enc}_K(\cdot)$ and $\text{Dec}_K(\cdot)$. The decryption oracle either outputs a plaintext or an invalid message symbol $\perp$. $\mathcal{A}$ outputs a pair of messages M_0, M_1 of the same length.
3. A random bit $b \leftarrow \{0,1\}$ is chosen, and then a ciphertext $C \leftarrow \text{Enc}_K(M_b)$ is computed and given to $\mathcal{A}$. We call C the challenge ciphertext.
4. The adversary $\mathcal{A}$ continues to have oracle access to $\text{Enc}_K(\cdot)$ and $\text{Dec}_K(\cdot)$, but is not allowed to query the latter on the challenge ciphertext itself. Eventually, $\mathcal{A}$ outputs a bit b'.
5. The output of the experiment is defined to be 1 if $b' = b$, and 0 otherwise.

We say the symmetric encryption scheme Π has indistinguishable encryptions under chosen-ciphertext attack if for all probabilistic polynomial time adversaries $\mathcal{A}$ there exists a negligible function $\text{negl}(\cdot)$ such that

$$Pr[\text{Experiment outputs 1}] \leq \frac{1}{2} + \text{negl}(n)$$

Again we assume Π to be probabilistic and not deterministic for the same reason as with IND-CPA, that encrypting the same message twice outputs different ciphertexts.
The decryption oracle returns $\perp$ if the plaintext message does not have the correct structure, correct headers etc, in other words does not look like it had been encrypted by the oracle. This distinction will be useful for proving IND-CCA later on.

IND-CCA is a reasonably strong notion about confidentiality, keeping the contents of a ciphertext secret, but we will now look at the property of integrity, ensuring that messages cannot be tampered with by an adversary

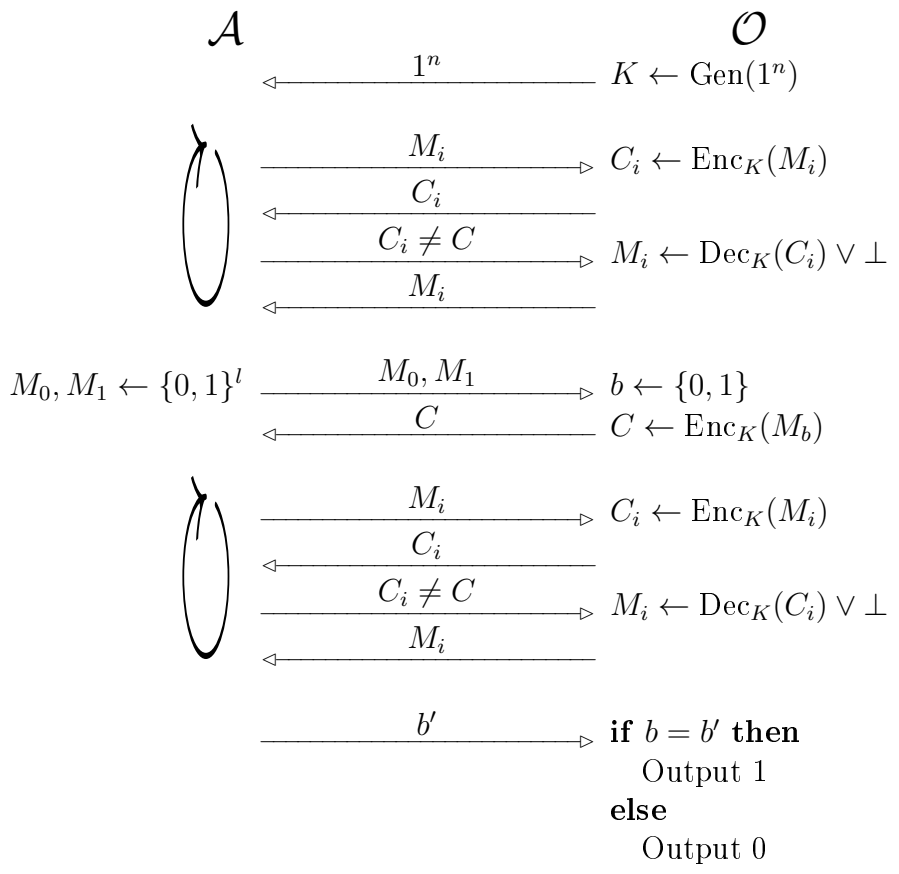

Figure 2.8: LOR-oracle IND-CCA game.

and still be accepted by their receiver. This property will also be useful for proving IND-CCA.

2.3.5 Definition: INT-PTXT

INT-PTXT is integrity of plaintexts. That is, the adversary's inability to forge a ciphertext that decrypts to a valid plaintext that has not been encrypted by the oracle before. By valid we mean that a plaintext message has the correct structure, correct headers etc, in other words a message that looks like it had been encrypted by the oracle. A LOR-oracle is not applicable in this setting as the adversary has to come up with his own answer rather than choose from predefined answers, so we use a regular oracle game.
For a symmetric key encryption system $\Pi \leftarrow$(Gen, Enc, Dec) with security parameter n:

1. A key K is generated by running $\text{Gen}(1^n)$.
2. The adversary $\mathcal{A}$ is given input 1^n and oracle access to $\text{Enc}_K(\cdot)$.
3. Querying $\text{Enc}_K(\cdot)$ with message M, the oracle returns $C \leftarrow \text{Enc}_K(M)$.
4. Eventually, $\mathcal{A}$ outputs a ciphertext C'.
5. The output of the experiment is defined to be 1 if C' decrypts to a valid message $M' \neq \perp$ that is distinct from all messages $\mathcal{M}$ queried to the encryption oracle.

We say the symmetric encryption scheme Π has plaintext integrity if for all probabilistic polynomial-time adversaries $\mathcal{A}$ there exists a negligible function $\text{negl}(\cdot)$ such that

$$Pr[\text{Experiment outputs 1}] \leq \text{negl}(n)$$

Unlike IND-CPA and IND-CCA we are not interested in a $\frac{1}{2} + \text{negl}(n)$ probability here because, as we mentioned earlier, this is not a LOR-oracle setting.

This notion is important because if a cryptosystem does not have INT-PTXT an adversary could inject a message of his own creation. However, it does not take into account that the adversary could create ciphertexts that are different from any seen before, but actually decrypt to a known plaintext, which could be exploited for replay attacks. This is why we have the notion of INT-CTXT.

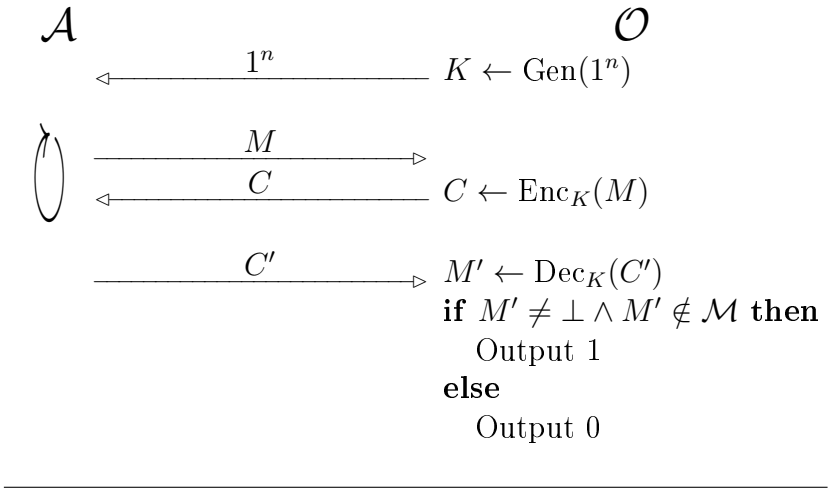

Figure 2.9: Oracle INT-PTXT game.

2.3.6 Definition: INT-CTXT

We have just looked at INT-PTXT which requires the decrypted plaintext to be different from any sent to the oracle. The difference with integrity of ciphertexts is that in this setting we require the forged ciphertext to be different from any output by the oracle. This is a stronger security notion than INT-PTXT as the forged ciphertext is allowed to decrypt to a known plaintext, and otherwise the two notions are identical.
For a symmetric key encryption system $\Pi \leftarrow$(Gen, Enc, Dec) with security parameter n:

1. A key K is generated by running $\text{Gen}(1^n)$.
2. The adversary $\mathcal{A}$ is given input 1^n and oracle access to $\text{Enc}_K(\cdot)$.
3. Querying $\text{Enc}_K(\cdot)$ with message M, the oracle returns $C \leftarrow \text{Enc}_K(M)$.
4. Eventually, $\mathcal{A}$ outputs a ciphertext C'.
5. The output of the experiment is defined to be 1 if C' decrypts to a valid message $M' \neq \perp$, and C' is distinct from all ciphertexts $\mathcal{C}$ returned by the encryption oracle.

We say the symmetric encryption scheme Π has ciphertext integrity if for all probabilistic polynomial time adversaries $\mathcal{A}$ there exists a negligible function $\text{negl}(\cdot)$ such that

$$Pr[\text{Experiment outputs 1}] \leq \text{negl}(n)$$

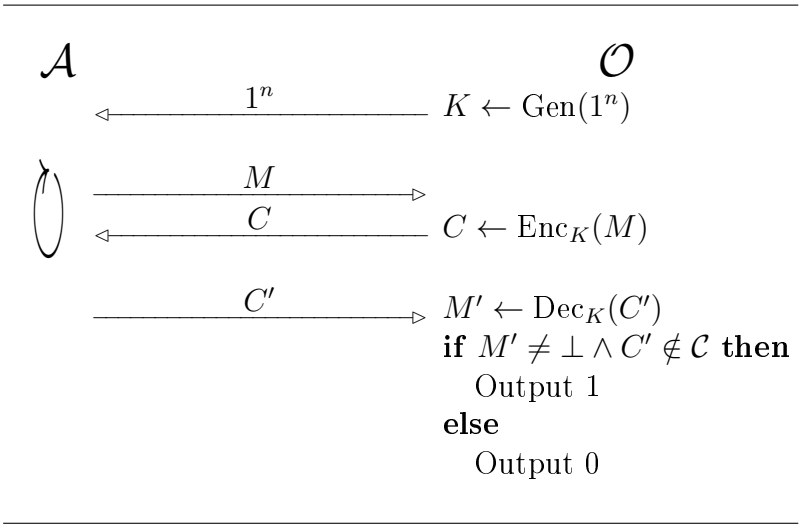

Figure 2.10: oracle INT-CTXT game.

The reason we are interested in having INT-CTXT is that it is a strictly stronger notion than INT-PTXT, which we will show in the following claim:

Claim 1: INT-CTXT is a strictly stronger notion than INT-PTXT, e.g.

$$\text{INT-CTXT} \Rightarrow \text{INT-PTXT}$$

Intuition Let's assume ¬INT-PTXT. Then there exists a probabilistic polynomial-time adversary that can forge a ciphertext that will decrypt to a plaintext never queried to the oracle. As decryption is deterministic, this implies the forged ciphertext is also different from all ciphertexts output by the oracle, which further implies ¬INT-CTXT. And by negation, INT-CTXT ⇒ INT-PTXT.

Furthermore, INT-CTXT can be used to prove IND-CCA. Together with IND-CPA it is proven to have IND-CCA.

Claim 2: From Paterson [15] we have that IND-CPA and INT-CTXT implies IND-CCA,

$$\text{IND-CPA} + \text{INT-CTXT} \Rightarrow \text{IND-CCA}$$

Intuition Assume Game 0 is the IND-CCA LOR-oracle game. Assume Game 1 is the same as Game 0 except decryption queries to the oracle will always return invalid message.
We see that the IND-CPA LOR-oracle game can perfectly simulate Game 1 because always returning invalid is equivalent to not having decryption access at all.
And if the cryptosystem has INT-CTXT then Game 1 is equal to Game 0 because if the adversary is unable to choose a new ciphertext that decrypts to a valid message, then the decryption oracle will not help him learn anything more than the messages he already sent to the encryption oracle.
And so we have that IND-CPA + INT-CTXT $\Rightarrow$ IND-CCA.
Note that this does not hold if the decryption oracle can return more than one error message.

So now we have established a strong notion for confidentiality as well as shown it can be provided by combining IND-CPA and INT-CTXT. In practice this is typically provided by using a message authentication code (MAC). This is a tag t derived from a message M using the function $t \leftarrow \mathrm{Mac}_K(M)$ with a shared secret key K. This tag is sent along with the message, and the receiving user can then use a verification function $\mathrm{Vrfy}_K(t, M)$ which outputs 1 if the tag is valid for the message and 0 if not. Or formally:

2.3.7 Definition: Message Authentication Code

We have from Katz et al. [13]: a message authentication code (or MAC) is a tuple of probabilistic polynomial-time algorithms (Gen,Mac,Vrfy) such that:

1. The key generation algorithm Gen takes as input the security parameter 1^n and outputs a key K with $|K| \geq b$.

2. The tag-generation algorithm Mac takes as input a key K and a message $M \in \{0,1\}^*$, and outputs a tag t. Since this algorithm may be randomized, we write this as $t \leftarrow \mathrm{Mac}_K(M)$.

3. The verification algorithm Vrfy takes as input a key K, a message M, and a tag t. It outputs a bit b, with $b = 1$ meaning valid and $b = 0$ meaning invalid. We assume without loss of generality that Vrfy is deterministic, and so write this as $b \leftarrow \mathrm{Vrfy}_K(M, t)$.

It is required that for every n, every key K output by $\mathrm{Gen}(1^n)$, and every $M \in \{0,1\}^*$, it holds that $\mathrm{Vrfy}_K(M, \mathrm{Mac}_K(M)) = 1$.

Naturally we require these MACs to be secure. By secure we mean that a probabilistic polynomial-time adversary could not forge a valid tag without knowing K. This is formalized in an oracle game:
Consider the following experiment for a message authentication code $\Pi \leftarrow$ (Gen,Mac,Vrfy), an adversary $\mathcal{A}$, and a value n for the security parameter:

1. A random key K is generated by running $\text{Gen}(1^n)$.
2. The adversary $\mathcal{A}$ is given input 1^n and oracle access to $\text{Mac}_K(\cdot)$. The adversary eventually outputs a pair (M, t). Let $\mathcal{T}$ denote the set of all MAC tags output by the oracle.
3. The output of the experiment is defined to be 1 if and only if **(1)** $\text{Vrfy}_K(M, t) = 1$ and **(2)** $t \notin \mathcal{T}$.

2.3.8 Definition: SUF-MAC

We say the message authentication code Π is strongly unforgeable under an adaptive chosen-message attack, or just secure, if for all probabilistic polynomial time adversaries $\mathcal{A}$ there exists a negligible function $negl(\cdot)$ such that

$$Pr[\text{Experiment outputs 1}] \leq negl(n)$$

As we have seen, using a secure MAC to provide data integrity with an IND-CPA secure encryption scheme can provide the strong confidentiality notion IND-CCA, but we would like to have a single notion that guarantees both confidentiality and integrity, which leads us to Authenticated Encryption.

Note that there also exists the notion of (weakly) unforgeable MACs for which the tag output by $\mathcal{A}$ is only required to match a message not queried to the oracle, but we will not cover it in this work as Bellare's article [5] shows that it will not help us guarantee Authenticated Encryption.

2.3.9 Authenticated Encryption

While IND-CCA is a strong notion for guaranteeing confidentiality, we would like to cover both confidentiality and data integrity under a single definition. This is known as Authenticated Encryption (AE), which is commonly achieved by combining secure encryption with a strongly unforgeable message authentication code. We have from Bellare's article [5] that AE is obtained by having both IND-CPA and INT-CTXT, e.g.:

$$\text{AE} \Leftrightarrow \text{IND-CPA} + \text{INT-CTXT}$$

It is tempting to think that any combination of secure encryption and message authentication will work, but as we will see the combination of even the best tools have turned out to yield an insecure result. We will consider three common approaches where K_1 is an encryption key, $\text{Mac}_K(\cdot)$ is a message authentication tag function outputting a tag t, and K_2 is a secondary Mac key:

- **Encrypt-and-authenticate**: A message authentication tag is computed from the message and the message is encrypted separately.

  $$C \leftarrow \text{Enc}_{K_1}(M) \text{ and } t \leftarrow \text{Mac}_{K_2}(M)$$

  The message receiver decrypts C to get M and then verifies t on M.

- **Authenticate-then-encrypt**: A message authentication tag is computed from the message, and the two are encrypted together.

  $$t \leftarrow \text{Mac}_{K_2}(M) \text{ and } C \leftarrow \text{Enc}_{K_1}(M||t)$$

  The receiver decrypts C to get M and t, and then verifies t on M.

- **Encrypt-then-authenticate**: The message is encrypted, and a message authentication tag is computed from the ciphertext.

  $$C \leftarrow \text{Enc}_{K_1}(M) \text{ and } t \leftarrow \text{Mac}_{K_2}(C)$$

  The receiver verifies t on C and if valid decrypts C to get M.

We will consider these insecure if even a single counterexample exists where the construction is insecure assuming the encryption scheme is IND-CPA and the MAC is a strongly unforgeable message authentication code.

Encrypt-and-authenticate From Katz et al. [13], let us note that an authentication code can be secure without having any confidentiality, and thus may leak the entire plaintext message. This is easy to see. Consider a secure MAC scheme Π = (Gen,Mac,Vrfy). Then the following is also secure $\text{Mac}'_K(M) = (M, \text{Mac}_K(M))$. This clearly leaks the entire plaintext and breaks confidentiality. Therefore, we do not consider encrypt-and-authenticate to be secure, even if there exists combinations for which it can be secure.

Authenticate-then-encrypt There also exists a counterexample for this construction, as seen in Katz et al. [13]. Consider the following encryption scheme:

- The function Transform(M) takes as input a bit string M of arbitrary length. Any 0 in M is transformed into 00, and any 1 is arbitrarily transformed into 01 or 10.
  The inverse function, Transform^{-1}, maps 00 back to 0 and 01 or 10 to 1. If 11 is encountered, the result is $\perp$. To clarify, Transform$^{-1}(0110) = 11$ and Transform$^{-1}(0111) = \perp$.

- The encryption function for this scheme is defined as $\text{Enc}_K(M) = \text{Enc}'_K(\text{Transform}(M))$, where Enc'_K is AES in CTR mode, which is IND-CPA secure. The important point is that Enc'_K generated a pseudorandom key stream which it XORs with the plaintext.
  Similarly, the decryption function is defined as $\text{Dec}_K(C) = \text{Transform}^{-1}(\text{Dec}'_K(C))$.

Now consider the following chosen-ciphertext attack:
An adversary receives a challenge ciphertext

$$C = \text{Enc}'_{K_1}(\text{Transform}(M||\text{Mac}_{K_2}(M))$$

Now the adversary simply flips the first two bits of the second block (note that the first block of counter mode is the nonce). He queries the decryption oracle with this new C', which is different from C. We now observe that if the first bit of the plaintext M was 1, then the decryption of C' will be valid as the transformed message was flipped from 01 to 10 or vice versa. And since the underlying message is unchanged, the MAC will also verify without issue.
If the first bit of M was 0 however, the oracle will return $\perp$ because the transformed message was flipped from 00 to the illegal 11. This means the adversary has now learned the first bit of the plaintext, and he can repeat this to uncover the full plaintext one bit at a time.
And so we see that this construction can be insecure in some cases, and we will therefore consider it insecure.

Encrypt-then-authenticate For this construction we cannot come up with a counterexample to break it, but we can give an intuitive proof that this is in fact secure for all combinations.
It is clear that this construction has ciphertext integrity since the secure MAC is computed from the ciphertext, which also means the MAC will not leak any information about the plaintext. And we have assumed that the

encryption scheme is IND-CPA secure, so we have INT-CTXT and IND-CPA. As we saw in Bellare's article [5], this guarantees AE. This is why encrypt-then-authenticate is the generally recommended construction.

2.4 Hash functions

A hash function is a function that takes input of arbitrary, or sometimes fixed, bit length and compresses it into a fixed length output, a digest. A common use for hash functions is integrity checking, verifying that a large bit string has not been tampered with or corrupted, by computing its digest and comparing this to one computed while the string was untouched. This way we do not have to compare everything bit-by-bit, which is an efficient way of validating large files after transfer, or to ensure a message received was not tampered with.

2.4.1 Definition: Hash function

From Katz et al. [13]: a hash function is a pair of probabilistic polynomial time algorithms (Gen,H) satisfying the following:

- Gen is a probabilistic algorithm which takes as input a security parameter 1^n and outputs a key s. We assume that 1^n is implicit in s.
- There exists a polynomial l such that H takes as input a key s and a string $x \in \{0,1\}^*$ and outputs a string $\mathrm{H}^s(x) \in \{0,1\}^{l(n)}$ (where n is the value of the security parameter implicit in s).

If H^s is defined only for inputs $x \in \{0,1\}^{l'(n)}$ and $l'(n) > l(n)$, then we say that (Gen, H) is a fixed-length hash function for inputs of length $l'(n)$.

In the fixed length case we require that $l' > l$ so that the hash function compresses the input.

For cryptographic hash functions we want certain properties. First off, a cryptographic hash function should be one-way, that is given a digest it is extremely difficult to find an input that hashes to this output. Secondly, it should be extremely unlikely that two inputs have the same digest. This is also known as collision resistance, which we will formally define.

2.4.2 Definition: Collision resistance

The strongest security notion for hash functions is collision resistance. This is the setting where an adversary $\mathcal{A}$ must find any two bit strings that have identical digests, with no parameters fixed beforehand. From Katz et al. [13]:

For a hash function Π =(Gen, H), an adversary $\mathcal{A}$, and a security parameter n:

1. A key s is generated by running Gen(1^n).
2. The adversary $\mathcal{A}$ is given s and outputs x, x'. (If Π is a fixed length hash function for inputs of length $l'(n)$ then we require $x, x' \in \{0,1\}^{l'(n)}$.)
3. The output of the experiment is defined to be 1 if and only if $x \neq x'$ and $\mathrm{H}^s(x) = \mathrm{H}^s(x')$. In such a case we say that $\mathcal{A}$ has found a collision.

We say the hash function Π is collision resistant if for all probabilistic polynomial-time adversaries $\mathcal{A}$ there exists a negligible function negl$(\cdot)$ such that

$$Pr[\text{Experiment outputs 1}] \leq \text{negl}(n)$$

Cryptographic hash functions are generally compared against this notion as it is the strongest, and having the property of collision resistance implies resistance in all other settings.

2.4.3 Weaker notions of security for hash functions

Collision resistance is the strongest security definition for hash functions. Strong in the sense that the hash function is assumed to be resistant even when the adversary has the most freedom to find a collision, but in some cases a weaker notion will suffice. Here we will consider three levels of security:

1. **Collision resistance**: This is the strongest notion as described above. No parameters are fixed, and the adversary has full freedom to find any two inputs that have identical digests.
2. **Second pre-image resistance**: A hash function is second pre-image resistant if given s and x it is infeasible for a probabilistic polynomial-time adversary to find $x' \neq x$ such that $\mathrm{H}^s(x') = \mathrm{H}^s(x)$. The adversary is more restricted in that the one hash function input is fixed beforehand, and if one can find a collision in this case it is easy to see that collision resistance is also broken.

3. **Pre-image resistance**: A hash function is pre-image resistant if given s and $y \leftarrow \mathrm{H}^s(x)$ (but not x itself) for a randomly chosen x, it is infeasible for a probabilistic polynomial-time adversary to find a value x' such that $\mathrm{H}^s(x') = y$. This is equivalent to reversing the hash function, and so we say that a cryptographic hash function should be one-way.

We will also consider a fourth, different notion, namely identical-prefix collision resistance:

- **Identical-prefix resistance**: A hash function is identical-prefix resistant if given s and prefix p it is infeasible for a probabilistic polynomial-time adversary to find x, x' such that $x \neq x'$ and $\mathrm{H}^s(p||x') = \mathrm{H}^s(p||x)$.

The important point about identical-prefix resistance is that if you can break this then you have also broken collision resistance, e.g.:

$$\neg\text{Identical-prefix resistance} \Rightarrow \neg\text{Collision resistance}$$

It is however not comparable to pre-image or second pre-image, and thus not stronger or weaker than either. The relationship between these four notions is shown in figure 2.11, in which a circle represents the domain of instances that are secure under a certain notion. These weaker notions will come in handy in the attack in section 6.2.3.

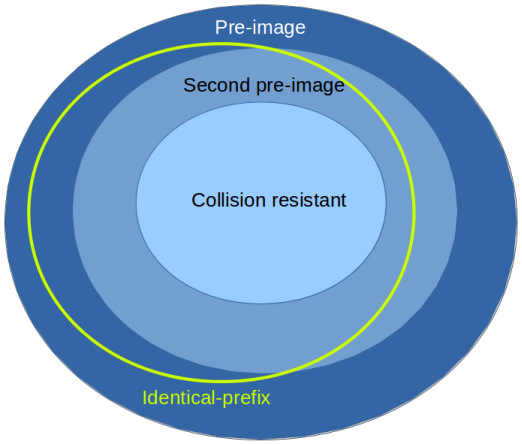

Figure 2.11: Relationship between hash function resistance notions.

2.4.4 Breaking hash functions

Finding a hash collision can always be brute-forced in $2^{n/2}$ evaluations, n being the number of bits output, because of the *birthday paradox*. A hash

function is said to be cryptographically broken if there exists an attack that can find a collision with expected evaluations less than $2^{n/2}$.

2.4.5 Secure Hash Algorithm 1 (SHA1)

SHA1 is an aging cryptographic hash function first published in 1993, taking input of arbitrary length and outputting 160 bit pseudo-randomness. There exists an identical-prefix collision attack better than 2^{80} operations as described in section 6.1.2, rendering SHA1 cryptographically broken. Using SHA1 today is considered bad practice for this reason.

Chapter 3

Protocols

In this chapter we will take a close look at the protocols and algorithms used in Telegram, and more specifically in their MTProto encryption scheme. We will look at how a client is first registered with the server, how it establishes a secure, end-to-end encrypted connection to another peer and how the internals such as the key derivation function are constructed.
This analysis is based on the official Telegram client source code for version 2.7.0, downloaded in mid April 2015 from github [22].

3.1 Device registration

Upon first installing the Telegram messaging application on a new device the user will be prompted to enter his/her phone number. This is used for authentication of the user for future logins as opposed to a username/email and password. After entering the phone number, the user will receive a five digit code by SMS in order to verify the number. Upon entering this code in the Telegram application, it will start the device authorization protocol.

This protocol goes as follows for a client $\mathcal{C}$ registering with server $\mathcal{S}$:

1. $\mathcal{C}$ sends a request to $\mathcal{S}$ with a random bit-string *nonce*.
2. $\mathcal{S}$ responds with another random bit-string *server_nonce*, a composite number n, and the fingerprint of a public RSA key.
3. $\mathcal{C}$ decomposes n into primes p and q such that $p < q$, which will serve as proof of work. $\mathcal{C}$ has a list of public keys stored locally, and selects the key $server_{pk}$ that matches the received fingerprint.

4. $\mathcal{C}$ chooses another random bit-string new_nonce, that unlike $nonce$ and $server\_nonce$ will not be sent in plaintext. $\mathcal{C}$ creates a payload of the three nonces and the numbers n, p and q. This payload, along with a digest of it, is encrypted using RSA under $server_{pk}$ and sent to $\mathcal{S}$.

5. $\mathcal{S}$ responds with Diffie-Hellman parameters g, p, g_a encrypted with AES-256 in IGE mode, using temporary AES key and AES IV derived from $server\_nonce$ and new_nonce.

6. $\mathcal{C}$ now chooses private value b and computes $g_b \leftarrow g^b \bmod p$ and $auth\_key \leftarrow (g_a)^b \bmod p$. g_b is sent to the server encrypted with AES-256 in IGE mode using the temporary key and IV.

The random bit strings $nonce$ and $server\_nonce$ are both 128-bit chosen at random by the client and server respectively. The number new_nonce is a 256-bit bit string chosen at random by the client.
n is a composite 64-bit integer, which is the product of two odd primes p and q, and the public RSA key fingerprint is the 64 least significant bits of the SHA1 digest of the server's public key.
The RSA encryption is performed using default Java RSA from javax.crypto.Cipher, which uses the standard PKCS #1 approach. In this approach the padding is pseudorandom data according to the Internet Engineering Task Force [8].

In order to decrypt the server response and obtain the DH parameters, the client must derive the 256-bit AES key and IV, as shown in figure 3.1. The decrypted response contains $nonce$, $server\_nonce$, the DH parameters g, p, g_a and a server timestamp used for synchronization. It also contains a 160-bit SHA1 digest of contents. The response is padded with randomness which is discarded with no checking apart from the length, which must be 0-15 bytes.

The client checks that the response contents match the digest that was sent with it, p must be a safe prime, meaning $q = \frac{p-1}{2}$ is prime, and $2^{2047} < p < 2^{2048}$. g is equal to 2, 3, 4, 5, 6 or 7 and it must hold that g generates a cyclic subgroup of prime order $\frac{p-1}{2}$, and that $1 < g_a < p - 1$, and if any of these checks fail, the protocol is aborted.

The private value b is a randomly generated 2048-bit bit string. The request that is sent to the server contains $nonce$, $server\_nonce$ and g_b, as well as a 160-bit SHA1 digest of these. The request is padded with randomness prior to encryption, and encrypted using the temporary AES key and IV.
The client also computes the shared long-term key $auth\_key \leftarrow g_a^b \bmod p$.

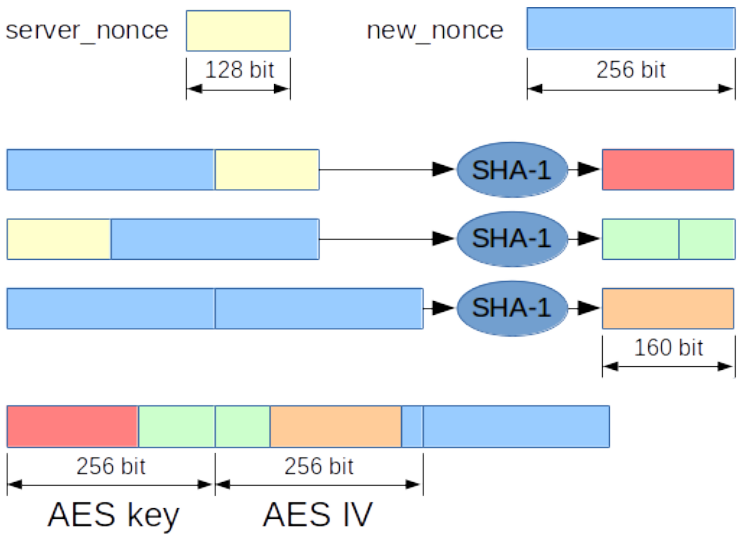

Figure 3.1: Deriving temporary AES IGE key and IV during registration.

Upon receiving g_b, the server does the same and sends response ok or failure if *nonce*, *server_nonce* or any DH parameters are wrong.

Client and server now have a shared secret key which is used only for client-server communication and regular (not end-to-end encrypted) chats, and the client device is now registered and authorized. The Telegram application will upload the client's phone contacts to the server in order to check if any of them have already registered, and if so it automatically adds them as Telegram contacts.

If the user tries to sign in from a new, unauthorized device, a new verification code will be sent. At first through the Telegram application on an authorized device, or through SMS if the former fails, and the above protocol repeats.

3.2 Key exchange

In this section we will take a look at how two users initiate an end-to-end encrypted secret chat.

According to Telegrams's website [24], a standard Diffie-Hellman (DH) key exchange is performed in order to have the users agree on a shared master secret, which later will be used for key derivation.

Two users A and B want to initiate a secret chat:

1. User A contacts the server in order to get the DH parameters, a prime p and a generator g. A also receives a salt to help his client generate a random secret value a.

2. A now computes public DH value $g_a \leftarrow g^a \bmod p$, and sends this to a second user B.

3. B now receives this chat request and accepts on only one of his authorized devices. He contacts the server to get the latest DH parameters and generates his secret value b.

4. B computes public value $g_b \leftarrow g^b \bmod p$ and sends it to A.

5. Both users now compute the shared secret $auth\_key \leftarrow g_a^b \bmod p = g_b^a \bmod p$, and the exchange is complete.

The clients are to verify that the DH parameters returned by the server satisfy the following: p must be a safe prime, meaning $q = \frac{p-1}{2}$ is prime, and $2^{2047} < p < 2^{2048}$. g is equal to 2, 3, 4, 5, 6 or 7 and it must hold that g generates a cyclic subgroup of prime order $\frac{p-1}{2}$. Also note that the DH parameters obtained from the server are fixed, but can be updated between application versions.

Secret values a, b are generated the following way:

$$a = r_{client} \oplus r_{server}$$

Where r_{client} is 2048 random bits generated by the client, and r_{server} is 2048 random bits generated by the server. The reason for this is to help clients with weak/flawed random number generators[1].

Clients should verify that $1 < g_a, g_b < p - 1$, and it is also recommended to check that $2^{2048-64} < g_a, g_b < p - 2^{2048-64}$. The probability of this is roughly 2^{-19}, or one in ten trillion[2] which we consider outside of the realm of possibility, and something must have gone wrong if it happens. If g_a or g_b is less than 2048 bits long, they are padded with zeros.

Diffie-Hellman key exchange is based on the discrete logarithm problem, and ensures that no passive eavesdropper could have learned $auth\_key$. It is

[1] Which was the case for Android back in August 2013 [1].

[2] Or ten US quintillion.

however susceptible to an active man-in-the-middle attack as basic Diffie-Hellman has no authenticity. We will look more into this in section 6.2.2. What Telegram does to guarantee authenticity is to give each user a visualization, a so called fingerprint, of $auth\_key$ and have them compare these.

The fingerprint of the shared secret key is the 128 least significant bits of SHA1($auth\_key$). This is visualized within the Telegram application as an 8x8 grid, each cell having one of four colors, as seen in figure 3.2. Users

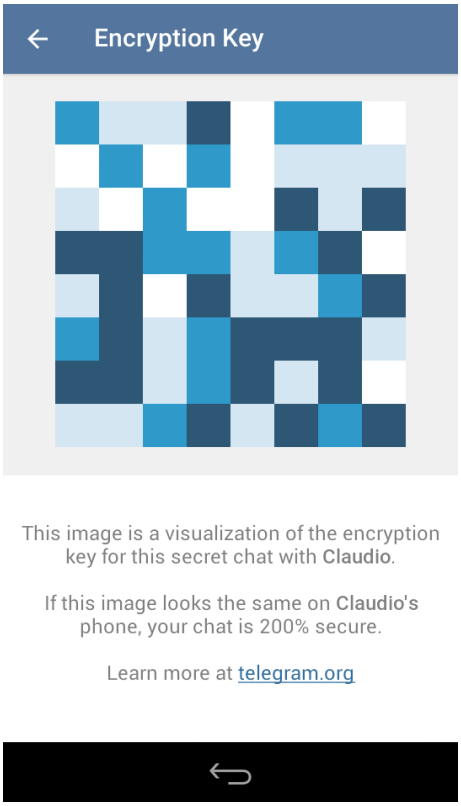

Figure 3.2: 128-bit key fingerprint visualization within Telegram for android.

are intended to meet in person and compare these, ensuring there are no differences. In this case, the chat session has not been compromised by a man-in-the-middle. But meeting in person often defeats the purpose of the chat, to communicate from afar, and is much more cumbersome than say, checking the users public key as with public key cryptography. This leads most users to simply take a screenshot of the key fingerprint and send this in the newly instantiated, unauthenticated chat. The man-in-the-middle has a key fingerprint for each user, and it is very easy for him to fool them by replacing the screenshot with one of his own. But assuming the users fol-

low the protocol, their chat session can be considered authenticated and safe from third parties.

3.3 Message encryption

Message encryption is where Telegram gets interesting. It uses its own encryption scheme, MTProto, which is intended to be computationally fast on mobile devices, and with no compromise on security. This is accomplished by using older primitives from back when computers were not as powerful, and combining them in such a way that known attacks do not apply.

First let us look at the contents of a message, which we will refer to as the payload:

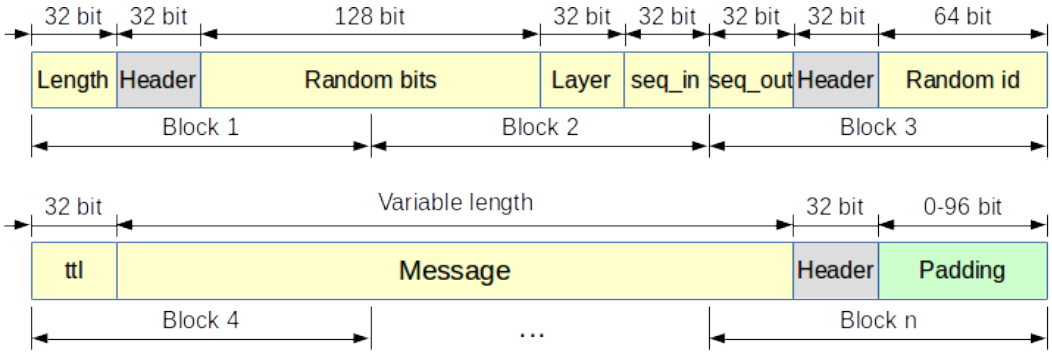

Figure 3.3: Contents of the payload to encrypt.

Keywords of payload contents of figure 3.3:

- **Length**: a 32 bit integer specifying the length of the payload (not counting padding or the length itself).
- **Header**: each payload contains three fixed headers related to the version of the protocol and the last one specifying the type of attached media, which we will always assume to be the *media_ empty* header.
- **Random_ bits**: 120 random bits generated by the sending client, and 8 bits used to specify the length of *random_ bits* in bytes. Used as a message salt.
- **Layer**: 32-bit integer specifying the version of the protocol.
- **seq_ in**: 32-bit counter for messages sent to the initiator of the chat.

- **seq_out**: 32-bit counter for messages sent from the initiator of the chat.
- **Random_id**: 64-bit random number that is also sent in plaintext, generated by the sending client.
- **ttl**: time to live, a 32-bit integer specifying the number of seconds the receiving user is allowed to see the message before it is deleted (a feature we will not cover in this work).
- **Message**: the message input by the user, of variable length.
- **Padding**: Not actually in the payload, padding is added just before encryption.

As of version 2.7.0 of April 2015, these are the contents of a message sent in secret chats. Prior to being sent in a secret chat, this payload is encrypted using MTProto as seen in figure 3.4.

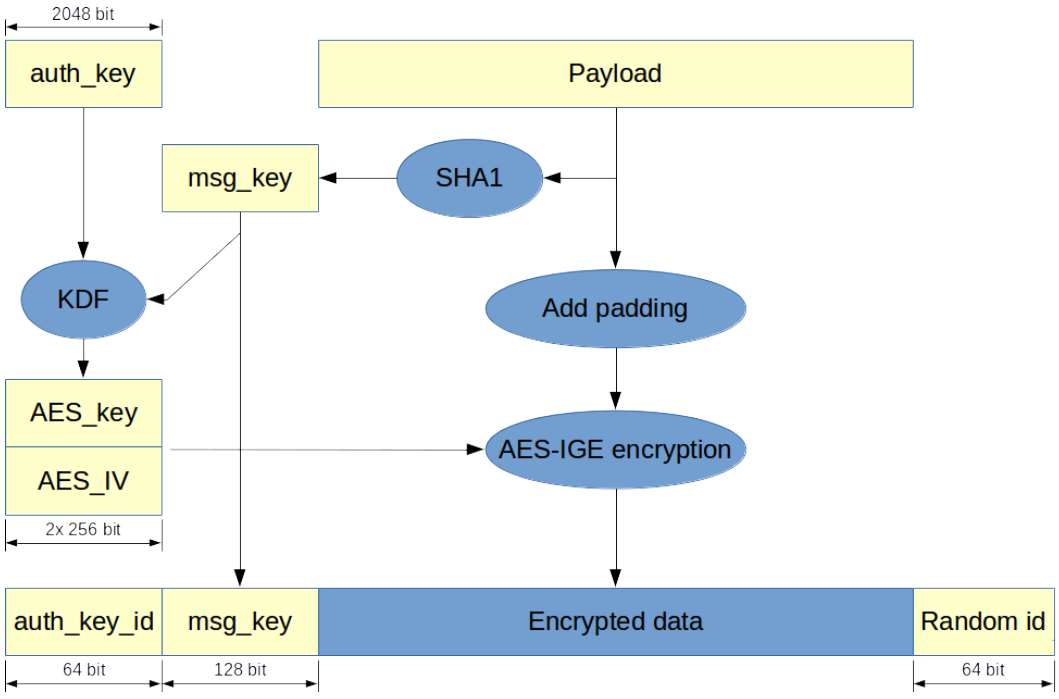

Figure 3.4: Flow of MTProto encryption scheme.

Keywords of MTProto encryption in figure 3.4

- **auth_key**: 2048-bit master secret, as exchanged in section 3.2. Only the 1024 most significant bits are used for key derivation.
- **Payload**: the message construction as seen in figure 3.3.
- **msg_key**: truncated 128 least significant bits of the SHA1 hash of the content to be encrypted. Used for integrity check.
- **Padding**: 0-96 random bits generated by the sending client, added to ensure every AES block has size 128 bit.
- **AES key and IV**: the 256 bit key and 256 bit IV required by AES in IGE mode. Derived using the KDF which we will see in section 3.4.
- **auth_key_id**: 64 lower-order bits of the SHA1 of the shared *auth_key*. In case of an *auth_key_id* collision, the shared key is regenerated.

As shown in figure 3.4, the integrity check *msg_key* is computed from everything but the padding. We will see later on that this differs from Telegram's own description in figure 4.1 where padding is included in *msg_key* and furthermore is said to be 0-15 bytes (0-120 bit).

The code snippet in figure 3.5 is from sending messages in secret chats. It hashes the payload and truncates it to find the message key, and afterwards adds padding.
In section 4.1.2 we'll see why the padding is in fact only 0-96 bits, and not 0-120 bits as claimed.

3.3.1 Message sequence numbers

A replay attack is an attack where an adversary records a sent message and sends it to the recipient again at a later time. This injects the message into the conversation without the original sender's knowledge. This is easily mitigated by including a message counter in every message, keeping track of the order messages appear in.

Similarly, a mirroring attack is when the adversary records a message sent by a user and sends it right back to him. If no precaution is taken, the message will be injected and the user will think the other client had sent a message identical to the one he just sent. Things get slightly trickier in this case, as using just a message counter might get fooled. The user cannot know for certain if the other client happened to send an identical message at the exact same time, in which case the message counters would be the same.

```
708         byte[] messageKeyFull = Utilities.computeSHA1(
                toEncrypt.buffer);
709         byte[] messageKey = new byte[16];
710         System.arraycopy(messageKeyFull, messageKeyFull.
                length - 16, messageKey, 0, 16);
711
712         MessageKeyData keyData = Utilities.
                generateMessageKeyData(chat.auth_key, messageKey,
                false);
713
714         len = toEncrypt.length();
715         int extraLen = len % 16 != 0 ? 16 - len % 16 : 0;
716         ByteBufferDesc dataForEncryption = BuffersStorage.
                getInstance().getFreeBuffer(len + extraLen);
717         toEncrypt.position(0);
718         dataForEncryption.writeRaw(toEncrypt);
719         if (extraLen != 0) {
720           byte[] b = new byte[extraLen];
721           Utilities.random.nextBytes(b);
722           dataForEncryption.writeRaw(b);
723         }
```

Figure 3.5: Padding added after deriving message key in implementation, as seen in SecretChatHelper.java from the official repository.

This is why MTProto uses two message counters as described on Telegram's website [28], one for incoming and one for outgoing messages (from the perspective of the initiator of the chat), and one bit of each is used to indicate which user had sent a given message. Assume we have two users A and B, A being the initiator of the chat.

- Each secret chat contains two counters, seq_out and seq_in.
- These are initialized to (0,0).
- Both are incremented by 1 after each sent message.
- They are transformed as follows:

$$2 * seq\_no + x$$

where x is determined by the following rule:

	seq_in	seq_out
message sent by A	0	1
message sent by B	1	0

Both clients maintain the raw (untransformed) counters, and the two transformed counters are included in every sent message.

Now if an adversary tries to do a replay attack, the message receiver will see that he already received a message with the same sequence number, and discard the replay message.
If the adversary attempts a mirroring attack on user A for example, A will see that the message is outgoing and not meant for him, and discard the message.

3.3.2 Message decryption

Message decryption in MTProto looks very similar to the encryption process, as shown in figure 3.6.

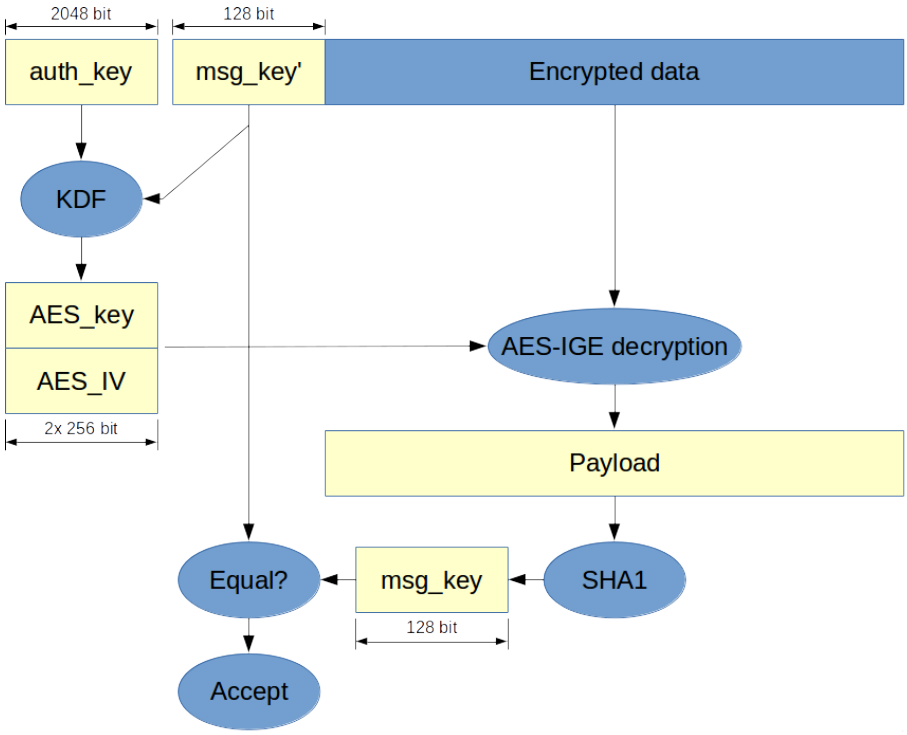

Figure 3.6: Flow of MTProto decryption scheme.

Not shown is that first the received *auth_key_id* matches the one associated with the secret chat on the client. If not, the message is discarded. We will go more in-depth with how parameters are checked upon decryption in algorithm 1.

The pseudocode in algorithm 1 takes as input $auth\_key\_id'$, msg_key', $encrypted\_data$. It uses methods SHA1($input$) and readBits($input, skip, length$) where $skip$ is the number of bits to skip and $length$ the number of bits to read from $input$. We will cover the KDF method in section 3.4.
At line 11 $payload$ is reduced in length, this is done to remove the padding before msg_key is computed.
We see that if at any time a check fails, the received message is discarded entirely with no error message returned.

Algorithm 1 MTProto decryption algorithm

Input: $auth\_key\_id', msg\_key', encrypted\_data$

```
1:  auth_key_id ← computeAuthKeyID(auth_key);
2:  if auth_key_id' ≠ auth_key_id then
3:      discard;
4:  end if
5:  AESkey, AESiv ← KDF(auth_key, msg_key');
6:  payload ← AES-IGE_Decrypt(encrypted_data, AESkey, AESiv);
7:  length ← readLength(payload);
8:  if length out of bounds then
9:      discard;
10: end if
11: payload ← removePadding(payload);
12: msg_key ← computeMsgKey(payload);
13: if msg_key' ≠ msg_key then
14:     discard;
15: end if
16: seq_in, seq_out ← readMessageCounters(payload);
17: if seq_in, seq_out ≠ local seq_in, seq_out  then
18:     discard;
19: end if
20: return payload;
```

3.3.3 Authenticity of encrypted messages

It is clear that MTProto provides confidentiality by using AES encryption, and plaintext integrity comes from the msg_key. Additionally, IGE mode provides ciphertext integrity by garbling the message if the ciphertext is tampered with. What is not as clear is how MTProto provides authenticity.

MTProto clearly does not use the encrypt-then-authenticate construction which we discussed in section 2.3.9, in fact it does not use a MAC at all. Instead, authenticity is verified by the receiver of a message by checking that the received msg_key matches the digest of the decrypted message. If this check passes, then the encryption and decryption keys must have been derived from the same shared secret, and thus it must have been encrypted and sent by the second user in a secret chat, who is authenticated by comparing the $auth\_key$ fingerprints.

3.4 Key Derivation Function

In this section we will look at how MTProto computes the encryption keys used for AES-IGE.

MTProto uses AES in IGE mode with 256 bit keys. IGE requires an initialization vector twice the size of one AES block, 256 bit. A new key and IV pair is generated for every message, and these are derived from the secret $auth\_key$ and the message itself. This is to ensure only the other holder of the $auth\_key$ can decrypt the message, and that the key will only decrypt the one message.

The way the key and IV is derived is by the use of multiple SHA1 hashes [27]. Presumably because a single SHA1 hash only outputs 160 of the 512 bits needed. Using a newer hash function such as SHA-512 would be ideal here, but MTProto has chosen to stick with SHA1.

As shown in figure 3.7, four SHA1 hashes are performed, each taking as input 256 bit of the $auth\_key$ and the 128 bit msg_key (which we recall is the truncated SHA1 digest of the message). In total, the 1024 most significant bits of the $auth\_key$ are used. Also note that the position of the msg_key in the hash input is shifted each time, presumably to make the outputs more "random" even though there is no evidence of this being true. The AES key and IV are then formed by taking substrings of each of the four SHA1 outputs and concatenating them to form two 256 bit strings. It is unclear why they are combined the way they are, and not just using 64 bit of each output per string.

The key derivation function is described more precisely in pseudo-code algorithm 3.8, using readBits($input, skip, length$) where $skip$ is the number of bits to skip and $length$ the number of bits to read from $input$.

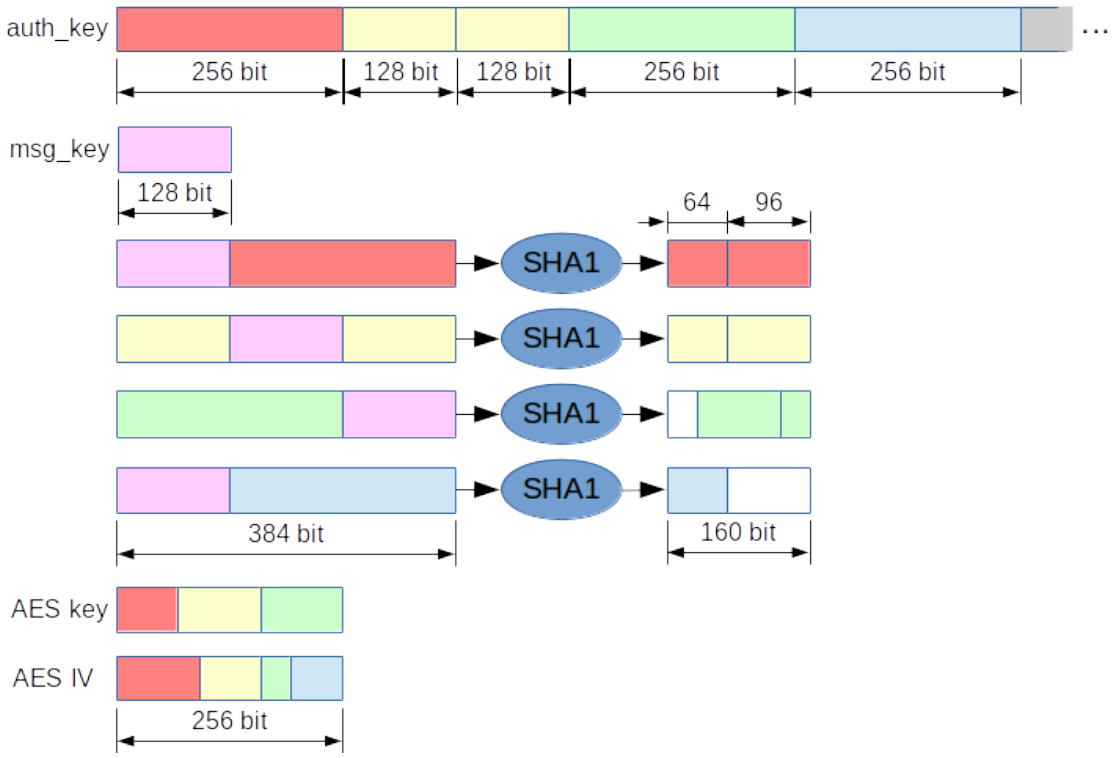

Figure 3.7: SHA1 based key derivation function used in MTProto.

```
1  sha1_a <- SHA1( msg_key || readBits(auth_key,0,256) )
2  sha1_b <- SHA1( readBits(auth_key,256,128) || msg_key ||
       readBits(auth_key,384,128) )
3  sha1_c <- SHA1( readBits(auth_key,512,256) || msg_key )
4  sha1_d <- SHA1( msg_key + readBits(auth_key,768,256) )
5  AES_key <- readBits(sha1_a,0,64) || readBits(sha1_b,64,96)
       || readBits(sha1_c,32,96)
6  AES_IV <- readBits(sha1_a,64,96) || readBits(sha1_b,0,64)
       || readBits(sha1_c,128,32) || readBits(sha1_d,0,64)
```

Figure 3.8: MTProto Key Derivation Function on input (auth_key,msg_key)

3.4.1 Forward Secrecy

Forward secrecy of a system is the property ensuring that a session key derived from a set of long-term keys will not be compromised if one of the long-term keys is compromised. Compromise of a single long-term key will only allow access to messages protected by that one key.

In MTProto [25] this corresponds to the AES keys being derived from the

auth_key which is fixed for a secret chat. If the *auth_key* is compromised, an adversary would gain access to every message sent. This is why version 20 of MTProto introduced a counter *key_use_count* keeping track of the number of times an *auth_key* is used to derive an AES key for encryption or decryption. If this counter reaches 100 or the secret key has been in use for more than one week, a new *auth_key* will be exchanged and the old one destroyed.

This new key exchange reuses the DH parameters that were received from the server during the initial key exchange, and no server randomness is added when generating new secret values a and b. One user will compute his public value g_a and send this to user B through the already established chat. B will respond with his new public value g_b, and both users compute and update the *auth_key* for the existing session, the same way that it was done in section 3.2. All older messages will remain decrypted and readable on both clients. The *auth_key* fingerprint used for clients to authenticate each other will not be changed.

This means that if an adversary gets hold of an old *auth_key* he will be able to derive AES keys and read 100 encrypted messages, but he will not be able to construct the next key from the public DH values.

3.4.2 Backwards compatibility

MTProto has received several updates over the years, and as a result there is no guarantee that all clients are on the latest version. To ensure clients can communicate across different versions, known as layers, the updated layers are made backwards compatible with the older ones.

The official Telegram client does a layer version negotiation as soon as the secret chat key exchange is completed. The *auth_key* is stored in an encryptedChat object along with a layer initialized to 0. The initiator of the chat A sends his layer in an encrypted message to B, and B responds with his layer version. Both clients store the lower of the two layers in the encryptedChat object in order to conform with the older version.

The current latest layer is version 23. Forward secrecy was introduced in layer 20, but apart from that no major changes have happened since layer 17. In layer 16 and down to 8 (the version that introduced secret chats), message sequence numbers were handled very differently. Rather than being included in each encrypted message, they were stored and maintained on the

proprietary server, and the message payload looked as shown in figure 3.9.

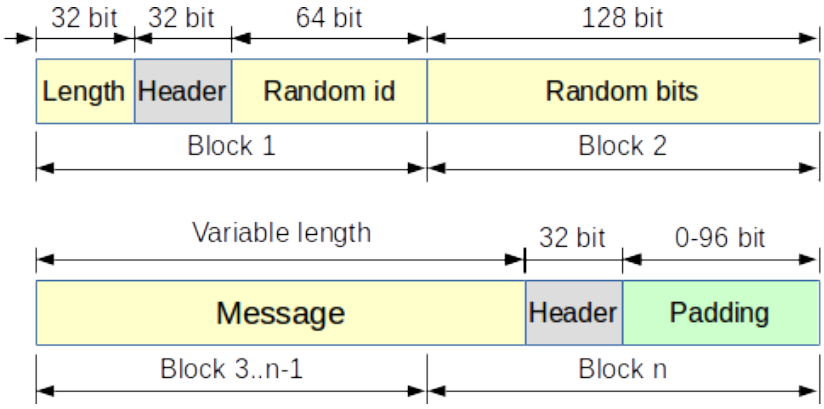

Figure 3.9: Contents of the payload to encrypt for versions below 17.

This meant that an encrypted messages did not contain any indication of the message's sequence number nor which of the two users had sent it, and the clients would have to trust the server to take care of this.

The messages themselves did not contain any mechanism to stop an unauthorized third party from sniffing sent messages and performing replay or mirroring attacks as discussed in section 3.3.1, which is mitigated by further encrypting the messages under the client-server authorization key from 3.1.

Chapter 4

Result of analysis

In the previous chapter we analyzed how MTProto is constructed and how it works. In this chapter we will outline weaknesses discovered and propose attacks that exploit them.

4.1 Random padding vulnerability

Under device registration and message encryption we observed that random padding is applied prior to AES-IGE encryption. This is not exploitable provided the padding has integrity and authenticity, which the diagram in figure 4.1 from Telegram's website [27] claims is the case. However as we saw in figure 3.4, the padding is not added until after the msg_key has been computed, and as a result the padding does not have integrity nor authenticity. This means the ciphertext block containing the padding could be modified undetected with non-negligible probability. Using this knowledge, we will present two attacks breaking IND-CCA and INT-CTXT, and by extension AE, focusing on the message encryption for MTProto.

4.1.1 IND-CCA attack #1: padding length extension

The following attack on MTProto exploits the fact that the length of the padding is never checked, and allows an adversary $\mathcal{A}$ to win the IND-CCA security game by adding extra blocks of padding to a ciphertext. The attack is visualized in figure 4.2.

1. $\mathcal{A}$ outputs messages M_0, M_1, $M_0 \neq M_1$ of the same length.
2. $\mathcal{O}$ chooses $b \in \{0, 1\}$ at random and outputs challenge ciphertext $C \leftarrow Enc_k(M_b)$.

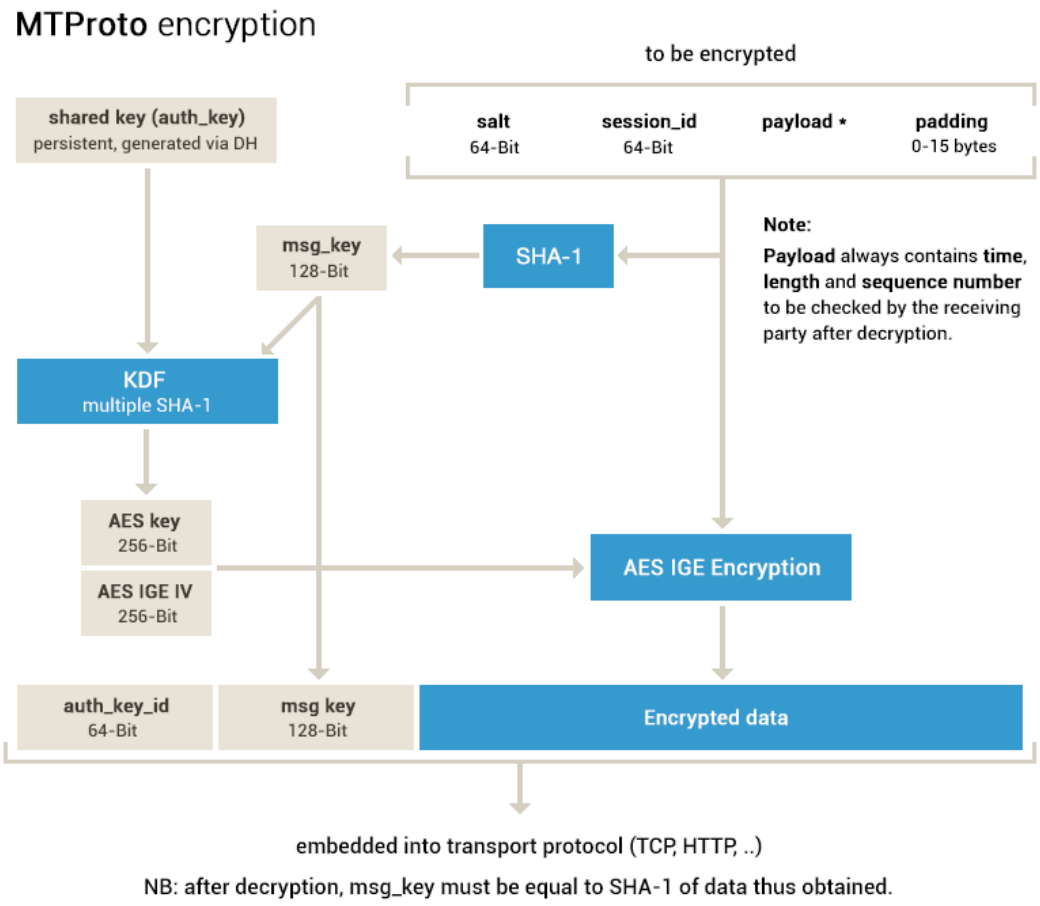

Figure 4.1: Flow of MTProto encryption scheme, from Telegram.org.

3. $\mathcal{A}$ now appends a 128 bit block of randomness c_r to C, asks O to decrypt $C' \leftarrow C||c_r \neq C$.

4. $\mathcal{O}$ decrypts C' and reads the payload length from it. Anything that comes after this length, e.g. the padding, is discarded, both the real padding and the extra block. $\mathcal{O}$ outputs $M' \leftarrow Dec_k(C||c_r) = M_b$.

5. $\mathcal{A}$ outputs 1 if $M' = M_1$ or 0 if $M' = M_0$.

This attack works for any number of extra blocks (at least 1), and the adversary wins with probability 1. This proves that MTProto does not have indistinguishability of encryptions under chosen-ciphertext attack. The same attack can be used to win the INT-CTXT game, just substitute the two messages M_0, M_1 with a single message M and the challenge ciphertext C will be the encryption of M. Again, the ciphertext C' is different from C, so it

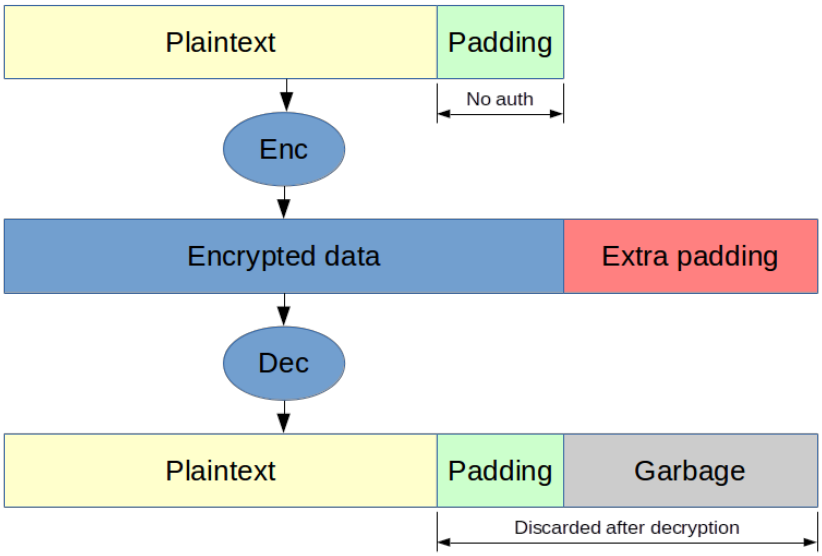

Figure 4.2: Modifying the ciphertext by extending the padding will not be detected upon decryption.

will be decrypted to a valid plaintext with probability 1.
We had from 2.3.9 that

$$\text{AE} \Leftrightarrow \text{IND-CPA} + \text{INT-CTXT}$$

and as we have shown that INT-CTXT is broken, that means AE is also broken.

Extending the message works fine, but it is in fact also possible to break IND-CCA and INT-CTXT without changing the length, with non-negligible probability.

4.1.2 IND-CCA attack #2: padding plaintext collision

The padding is never authenticated, so modifying (and garbling) the last 16-byte (128-bit) block has a non-negligible probability of having a collision in the plaintext bits, disregarding the padding. Consider the following:

1. $\mathcal{A}$ sends a message M, the length of M in bytes being equal 1 mod 16.
2. MTProto will hash M into $msg\_key \leftarrow \text{SHA1}(M)$ (truncated to 128 bits), which provides integrity for the plaintext.

3. Prior to encryption, 15 random bytes of padding r are added to the message, and the payload is encrypted

$$C \leftarrow Enc_k(M||r)$$

4. $\mathcal{A}$ now modifies the last 16 byte block of the ciphertext to get $C' \neq C$.

5. $\mathcal{A}$ outputs C'.

Upon decryption $M' \,||\, padding \leftarrow Dec_k(C')$, the last block will be garbled due to the non-malleability of IGE. However, since the 15 padding bytes are not part of msg_key, only the first byte has to match the original message when $msg\_key \leftarrow \text{SHA1}(M')$ is computed and checked against the received msg_key'. This means we have a $2^{-8} = \frac{1}{256}$ probability that the last byte of M' will be the same as that of M, and the modified ciphertext will be accepted without detection.

This would be the best case scenario for the attack. However, when the message String in Java is serialized, the result is a multiple of 4 bytes with any unused bytes made 0. Take for example the string 'Test'. The first byte is the string length, 4. Next are the four bytes corresponding to each character, followed by a message header from MTProto. We expect these to come in succession, but we actually get a three 0 byte padding before the header.

```
1 Expected: 04 54 65 73 74 4A 5C 9F 08
2 Reality : 04 54 65 73 74 00 00 00 4A 5C 9F 08
```

As we see in figure 3.3, all data in the payload is a multiple of 4 bytes, therefore the total message encrypted is a multiple of 4 bytes, and as such it's impossible to have a message length equal 1 mod 16 bytes. Instead, the best case is to have a length equal 4 mod 16, with 12 bytes of padding instead of 15, and so the probability of finding a plaintext collision is reduced from 2^{-8} to $2^{-32} = \frac{1}{4294967296}$, which is much less feasible.

The attack can be used to get the oracle to decrypt the challenge ciphertext in the IND-CCA security game and will also decrypt to a valid plaintext in the INT-CTXT game. As 2^{-32} is still non-negligible in the security parameter, we again conclude that MTProto does not have indistinguishability of ciphertexts nor ciphertext integrity, and by extension not authenticated encryption.

These attacks would both be impossible if the implementation followed figure 4.1 and included the padding in msg_key.

4.2 Replay and mirroring attacks in older versions

As we saw in section 3.4.2, in the still supported versions 16 and down the clients had to trust the server to maintain message sequence numbers as the messages themselves contained no such mechanism. But this means a malicious server is given full control of the flow of messages. The server, although incapable of reading message contents, can choose to withhold, replay, reorder or mirror messages at will. If the server was to somehow learn the contents of older messages, it could freely inject them at any time and impersonate either or both users. This vulnerability would be less significant if there was forward secrecy as the server would have at most 100 valid messages to inject at a time, but recall that forward secrecy was not introduced until version 20.

Do note however that the version negotiation is done after the key exchange has taken place, and as such the server isn't able to actively lower the negotiated version. One of the clients will have to actually be at version 16 or older for the chat to be vulnerable.

4.3 Timing attacks on MTProto

A timing attack is an attack based on the time between sending a message and receiving an error response. If a system checks several parameters sequentially and responds as soon as one check fails, it is then possible to figure out how many checks have passed based on the timing. Telegram is intended for use on mobile devices and one might argue that weak signal will influence timing by an order of magnitude more than CPU cycles, but disregarding this we will look into the implementation of MTProto and see if it would be susceptible to a timing attack.

Looking through the source code for Telegram's decryption implementation reveals that upon receiving a message, the following checks are performed in order:

1. Received *auth_key_id* is compared to the one stored locally.
2. After decryption the message length in bytes is checked to be larger than 0 and smaller than the number of received bytes.
3. The *msg_key* is computed and compared to the one sent in plaintext.

4. The sequence numbers are checked against the local counter.

The decryption process immediately aborts if any check fails, which led us to believe that a timing attack might be possible. This was put to the test by sending a message in a secret chat, but with the encrypted payload replaced with random bits.

The result is that when any check fails, the message is simply not accepted by the receiver and is forever marked as unread in the sending client, and the sender does not get any notification about failure. As such, it is infeasible to perform a remote timing attack on MTProto. However, if an attacker had gotten a user to install a malicious application on their device, this could gather information about when Telegram is running. But this is a whole different topic, and we will not go into further detail with it.

Chapter 5

Experimental validation

In this chapter we will implement and verify the attacks proposed in chapter 4. In order to verify these an MTProto simulator has been built, which functions as a fake server between two users. This is built by extracting the core functionality of the encrypt and decrypt methods from the Telegram for Android source code, stripping them of all actual network connectivity, into a simple Java program. Two fake users A and B are generated with random user IDs, and a secret chat is set up with a randomly generated authorization key, as opposed to running the Diffie-Hellman key exchange.
User A can then be instructed to encrypt a message, outputting the ciphertext to the fake server. The fake server can apply any of the proposed attacks before forwarding the modified ciphertext to user B. B will then use the decrypt method to verify that the message passes all checks and would have been accepted in an official client, or reject if the message is corrupt.

5.1 Attack #1: padding length extension

The attack from 4.1.1 has been implemented, and it works as follows. A method addPadding() is given two arguments: the encrypted message object containing the ciphertext and an integer specifying the number of blocks of padding to be appended.

A new byte array of size equal to the original ciphertext plus the extra blocks is created, and the ciphertext is copied into it. Next an array of random bytes is created and copied into the new array. The source code for this can be seen in figure 5.1.

Experiments have been run appending 1 to 20 blocks of extra padding, and in

```
1  public static ByteBufferDesc addPadding(TLRPC.
       TL_messages_sendEncrypted encrypted, int numBlocks) {
2    if(numBlocks > 0) {
3      int fakePaddingSize = 16 * numBlocks;
4      int length = encrypted.data.limit();
5      byte[] encryptedMessageExtended = new byte[length +
         fakePaddingSize];
6      encrypted.data.position(0);
7      System.arraycopy(encrypted.data.readData(length), 0,
         encryptedMessageExtended, 0, length);
8      byte[] fakePadding = new byte[fakePaddingSize];
9      Utilities.random.nextBytes(fakePadding);
10     System.arraycopy(fakePadding, 0,
         encryptedMessageExtended, length, fakePadding.length
         );
11     return new ByteBufferDesc(encryptedMessageExtended);
12   }
13   else return encrypted.data;
14 }
```

Figure 5.1: Code snippet for padding length extension attack

every case the modified ciphertext has been decrypted and accepted without errors. This confirms the effectiveness of the attack, and if used as proposed in section 4.1.1 this breaks IND-CCA security.

Short-term countermeasure This attack could be mitigated by simply checking the length of the padding upon decryption. The following code snippet is from the MTProto decryption implementation, where **is** is a byte array holding *auth_key_id*, *msg_key* and the encrypted payload, as output in 3.4. The payload is decrypted, and the payload length is read from the plaintext.

```
1319     int len = is.readInt32();
1320     if (len < 0 || len > is.limit() - 28) {
1321       return null;
1322     }
```

is.limit - 28 is the length of the payload contents with padding and **len** is the length without padding. The padding should never be longer than 15 bytes, and so what we need to do is add **is.limit() - 28** $>$ **len** $+$ **15**.
This mitigates the padding length extension attack and as it is only a conditional check, it has no impact on the encryption and decryption methods.

```
1        int len = is.readInt32();
2        if (len < 0 || len > is.limit() - 28 || is.limit() - 28
             > len + 15) {
3          return null;
4        }
```

This means it has no impact on client compatibility, and an updated client would have no trouble communicating with one that does not have this patch.

5.2 Attack #2: padding plaintext collision

The implementation of 4.1.2 goes as follows. The ciphertext is copied into a new byte array, except for the last 16 byte block. Now 16 random bytes are generated and copied in their place.

```
1  public static ByteBufferDesc changeLastBlock(TLRPC.
       TL_messages_sendEncrypted encrypted){
2    byte[] data = new byte[encrypted.data.limit()];
3    encrypted.data.position(0);
4    System.arraycopy(encrypted.data.readData(encrypted.data.
       limit()), 0, data, 0, encrypted.data.limit() - 16);
5    byte[] fakeData = new byte[16];
6    Utilities.random.nextBytes(fakeData);
7    System.arraycopy(fakeData, 0, data, encrypted.data.limit()
       - 16, fakeData.length);
8
9    return new ByteBufferDesc(data);
10 }
```

Figure 5.2: Code snippet for padding plaintext collision attack

With a $\frac{1}{2^{32}}$ probability of success, there's a $\frac{2^{32}-1}{2^{32}}$ probability of failure. To have a probability of .5 of finding a collision, one would have to do x attempts, for

$$1 - \left(\frac{2^{32}-1}{2^{32}}\right)^x = 0.5 \Rightarrow 1 - 0.5 = \left(\frac{2^{32}-1}{2^{32}}\right)^x$$

$$\begin{aligned} x &= \log_{\frac{2^{32}-1}{2^{32}}}(0.5) \\ &= 2.977 \times 10^9 \end{aligned}$$

One instance of the attack finished in 1.592×10^9 attempts, taking 5.8 hours without any parallelization on a Intel Xeon E5-2680 v2 @ 2.80GHz. Compar-

ing plaintexts, the last block of the original message and the found collision were

```
Original: 4A5C9F08 D4F33C82 657924AE 4B403ECB
Found   : 4A5C9F08 6255B92D 62F6AC8B 318374B0
```

For every pair of hexadecimal characters representing one byte, we see that the first four bytes are indeed identical, whereas the padding bytes are different.

Other tests have finished in 1.606×10^9 to 7.109×10^9 attempts. This probabilistic approach is trivially parallelizable as all attempts are fully independent, but we will not implement optimizations in this work.

Short-term countermeasure In order to mitigate this vulnerability, we need to verify integrity of the padding. This is easily accomplished by including the padding in the *msg_key*, which serves as integrity for the message already, and which would make secret chat message encryption function as described in figure 4.1 from Telegram's website. All the way back in figure 3.5 we saw how the *msg_key* is computed before padding is added, and simply rearranging these code chunks will give the following:

```
        len = toEncrypt.length();
        int extraLen = len % 16 != 0 ? 16 - len % 16 : 0;
        ByteBufferDesc dataForEncryption = BuffersStorage.
            getInstance().getFreeBuffer(len + extraLen);
        toEncrypt.position(0);
        dataForEncryption.writeRaw(toEncrypt);
        if (extraLen != 0) {
          byte[] b = new byte[extraLen];
          Utilities.random.nextBytes(b);
          dataForEncryption.writeRaw(b);
        }

        byte[] messageKeyFull = Utilities.computeSHA1(
            dataForEncryption.buffer);
        byte[] messageKey = new byte[16];
        System.arraycopy(messageKeyFull, messageKeyFull.
            length - 16, messageKey, 0, 16);

        MessageKeyData keyData = Utilities.
            generateMessageKeyData(chat.auth_key, messageKey,
            false);
```

Now the padding is part of the *msg_key*, and thus has integrity. We will

also have to modify the decryption implementation to include the padding when checking integrity, where the current implementation is the following code snippet:

```
1323        byte[] messageKeyFull = Utilities.computeSHA1(is.buffer
                , 24, Math.min(len + 4 + 24, is.buffer.limit()));
1324        if (!Utilities.arraysEquals(messageKey, 0,
                messageKeyFull, messageKeyFull.length - 16)) {
1325          return null;
1326        }
```

What has to be done here, is replacing **Math.min(len + 4 + 24, is.buffer.limit())** with just **is.buffer.limit()**.

```
1           byte[] messageKeyFull = Utilities.computeSHA1(is.buffer
                , 24, is.buffer.limit());
2           if (!Utilities.arraysEquals(messageKey, 0,
                messageKeyFull, messageKeyFull.length - 16)) {
3             return null;
4           }
```

Now we check the integrity of the padding, and the padding block plaintext collision attack now has probability 2^{-128} of succeeding, which is negligibly low in the length of an AES block.

Checking integrity of the padding also happens to mitigate the padding length extension attack, however this security patch will render updated clients incompatible with unupdated ones, as messages sent between them will always fail the integrity check.

5.3 Malicious server attacks

The attack described in section 4.2 was tested on the local test setup in the following experiments:

Two users A and B communicate through the malicious server S. A's layer is set to 16. The experiments are all based on S discarding or storing and later forwarding messages, and the concepts of the four attacks are shown in figure 5.3. Carrying out these attacks in our local test setup, we found that in all cases the clients will accept messages and the attacks are successful.

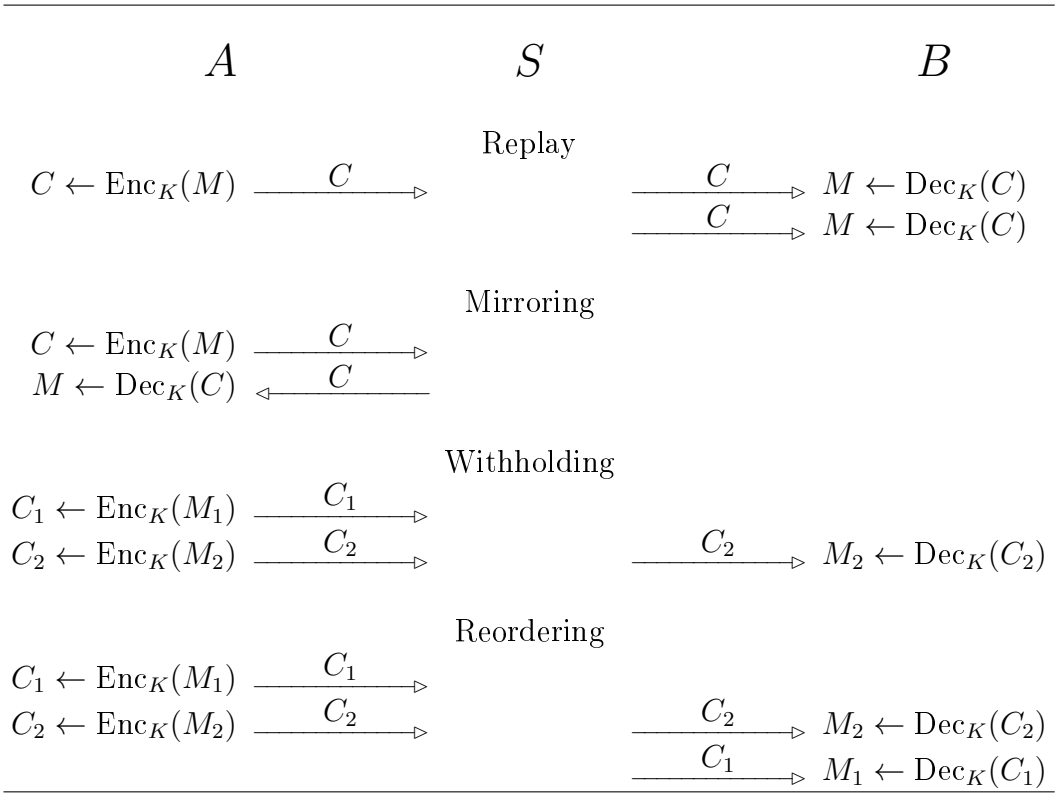

Figure 5.3: The naive MitM attack.

Short-term countermeasure The way to mitigate these attacks is of course to take away control from the server, and let the clients themselves handle message counters independently. This is exactly what was done in version 17, and so what needs to be done is to remove backwards compatibility for versions 16 and older.

Chapter 6

Known attacks

In this chapter we will outline known attacks on the primitives MTProto is built upon, as well as attacks on MTProto itself. Looking at SHA1, we will go in-depth with how much it would cost to fine a collision on today's hardware, and the attacks on MTProto are all man-in-the-middle attacks, one of which has been patched out.

6.1 Known attacks on primitives

6.1.1 IGE is not BACPA secure

Bard [4] states that IGE encryption mode is CPA secure, but it is proven to be blockwise-adaptive chosen-plaintext attack (BACPA, 2.3.3) insecure, for both the general and primitive notion.

Recall that in IGE encryption, a plaintext block is XOR'd with the previous ciphertext block prior to running the encryption function.

$$c_{i-1} \leftarrow \mathrm{f}_K(m_i \oplus c_{i-1}) \oplus m_{i-1}$$

To win the BACPA security game, the adversary needs to find a collision between two blocks before they enter the encryption function, after they are XOR'd with the precious ciphertexts. This way, the output ciphertext blocks will be the same, just XOR'ed with a plaintext block that the adversary already knows. Creating a collision is trivially accomplished due to the way IGE mode does XOR before and after encryption. All the adversary has to do is send two random message blocks to get their encryption, and then he can compute a third block that will encrypt to the same as the second block, causing a collision. This attack is also shown in figure 6.1.

1. A key K is generated by running $\text{Gen}(1^n)$ and 1^n is given to $\mathcal{O}$ and $\mathcal{A}$. $\mathcal{O}$ chooses two initial blocks m_0, c_0 at random.
2. $\mathcal{A}$ chooses and outputs two identical message blocks m_{01}, m_{11}.
3. $\mathcal{O}$ computes and returns
$$c_1 \leftarrow m_0 \oplus \text{Enc}_K(m_{b1} \oplus c_0)$$
4. $\mathcal{A}$ chooses and outputs two identical message blocks m_{02}, m_{12}.
5. $\mathcal{O}$ computes and returns
$$c_2 \leftarrow m_{b1} \oplus \text{Enc}_K(m_{b2} \oplus c_1)$$
6. $\mathcal{A}$ now computes
$$m_{03} \leftarrow m_{02} \oplus c_1 \oplus c_2$$
and chooses $m_{13} \neq m_{03}$.
7. O computes and returns
$$c_3 \leftarrow m_{b2} \oplus \text{Enc}_K(m_{b3} \oplus c_2)$$
8. $\mathcal{A}$ now checks if
$$m_{01} \oplus m_{02} \oplus c_3 \stackrel{?}{=} c_2$$
If this is true then $\mathcal{A}$ knows $b = 0$, else $b = 1$ and $\mathcal{A}$ has thus won the game.

This attack wins the BACPA oracle game with probability 1. The reason for the way m_{03} is chosen is that if $b = 0$, then:

$$\begin{aligned} c_3 =& m_{02} \oplus \text{Enc}_K((m_{02} \oplus c_1 \oplus \cancel{c_2}) \oplus \cancel{c_2}) \\ =& m_{02} \oplus \text{Enc}_K(m_{02} \oplus c_1) \end{aligned}$$

And in this case we can turn c_3 back into c_2:

$$\begin{aligned} m_{01} \oplus m_{02} \oplus c_3 =& m_{01} \oplus \cancel{m_{02}} \oplus \cancel{m_{02}} \oplus \text{Enc}_K(m_{02} \oplus c_1) \\ =& m_{01} \oplus \text{Enc}_K(m_{02} \oplus c_1) \\ =& c_2 \end{aligned}$$

If the result of the above is not c_2 then we can say for sure that $b = 1$.

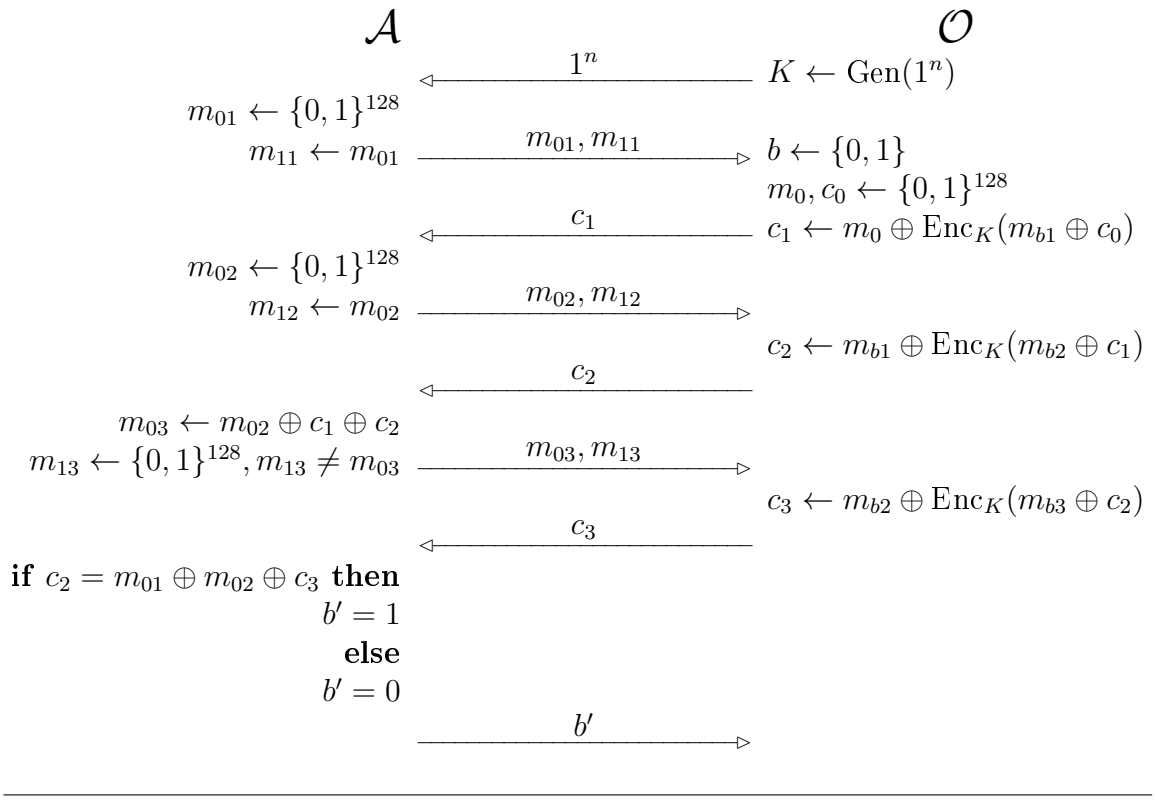

Figure 6.1: LOR-oracle IND-BACPA attack.

This means systems using IGE do not even have indistinguishable of encryptions under (blockwise-adaptive) chosen-plaintext attack, one of the weaker notions, and as only a single query was made for which $m_{1i} \neq m_{0i}$ it holds for both the general and primitive notion. But as Telegram points out [26], it does not apply to MTProto. This is because the AES key and IV are dependent on the message content, which has to be fixed prior to encryption. Any blockwise-adaptive attack is impossible because one cannot change a single block without changing the encryption key.

6.1.2 Security of SHA1

SHA1 is used in Telegram because it is less computationally intensive than most hash functions, even though better alternatives exist such as the more secure SHA2 and SHA3, or even BLAKE [3] which was one of the final candidates for SHA3 and performs faster than SHA1. SHA1 is cryptographically broken as a 2^{61} identical-prefix collision attack has been published by Stevens [21] in 2012, but how costly is it actually to break?

In 2012, Jesse Walker from Intel did a back-of-the-envelope prediction of the future cost of finding SHA1 hash collisions, as posted on Bruce Schneier's blog [19].

The short analysis, based on Stevens' 2^{60} attack[1], assumes the cost of finding a collision is 2^{74} cycles as each SHA1 operation requires 2^{14} cycles.

Jesse estimates that a modern server with 4 processors, each with 8 cores, is able to carry out 2^{61} cycles per year. Taking Moore's law into account, that number should have increased to 2^{63} by this year 2015, 2^{65} by 2018, and so on.

In 2012 it would have cost $2.77 million to carry out this attack on an Amazon rental server. Assuming the rental price is fixed as the server hardware gets upgraded, one could find a collision for $700k in 2015, $173k by 2018, and $43k by 2021. These lower costs are within reach of organized criminals and university research projects.

Looking at Amazon's offerings shows that these estimates are not entirely accurate. Jesse most likely looked at what is called the m1.small [2] which indeed costs $0.04 per hour or $350 per year. This only offers a single core and not four 8-core CPUs. Since then Amazon has upgraded to the newer

[1] Appears to be a typo in the analysis, Stevens' paper [21] reads 2^{61}.

Intel Sandy Bridge architecture and the price remains comparable, but it is not clear how big the performance gain is and hard to confirm the 2015 prediction.

It is more interesting to look at how the introduction to cryptocurrencies have changed the price of computing hashes, take for example Bitcoin which is based on computing SHA256 hashes. Looking at statistics for Bitcoin mining [7], we see that currently about 4×10^{17} hashes are performed every second by the community. This is roughly 2^{59} hashes per second, which means that if they all focused their computational power on finding one SHA1 collision this could be done in about 4 seconds.

If we look at the price of specialized hardware for solving SHA256 hashes [31], we find that the AntMiner S5 has the highest hash rate per power consumed at 1.15×10^{12} hashes per second at 590 W, and costing only $370. As $1.15 \times 10^{12} \approx 2^{40}$ it needs to run for

$$\frac{2^{61}}{2^{40}} = 2^{21} \text{ seconds}$$

or roughly 24 days to find a collision. The cost of running it for 24 days is

$$\frac{590 \cdot 24 \cdot 24}{1000} = 339.84 \text{ kWh}$$

which at $0.12 per kWh [20] comes out to about $40 per collision (plus $370 for the hardware).

Bear in mind this is for computing SHA256 hashes, not SHA1, but as SHA1 is less computationally intensive it would presumably be cheaper to build specialized hardware for computing it. This is shockingly cheap and could be afforded by anyone, and we might as well assume that a SHA1 collision could be trivially found. And let us have a look at the consequences if this was the case.

6.1.3 SHA1 collision consequences for Telegram

In MTProto, SHA1 is used for computing the *auth_key_id* fingerprint, the *msg_key*, and for the *auth_key* visualization. It is also used in the key derivation function, but as neither the input nor output ever leaves the client's device there is no way of finding a collision here.

- Finding an input that hashes to the *auth_key_id* is made significantly easier due to the fact that it is truncated from 160-bit to 64-bit. However, this requires a pre-image attack which is harder than a hash collision and Stevens' attack is not applicable here. Furthermore, finding this pre-image will not give the adversary any advantage unless it happens to match the 1024 bit of the secret *auth_key* used for key derivation.

  Assuming SHA1 hashes are uniformly distributed, if the adversary tried all 2^{1024} inputs he would find $2^{1024-64}$ collisions with no way of telling which is the right one, unless he knows a plaintext and its encryption. This is an astronomical number of values to search through and is not feasible to carry out in practice.

- Finding a collision M' for the truncated 128-bit *msg_key* of a given encrypted message M could potentially allow an attacker to forge a message. However, recall that the *msg_key* is the digest of the plaintext. This means that the adversary, when given the hash function and a ciphertext C, would have to be able to output a ciphertext C' of which the decryption is a pre-image such that

  $$\text{SHA1}(\text{Dec}_K(C)) = \text{SHA1}(\text{Dec}_K(C'))$$

  Telegram's FAQ [26] argues that this mitigates known attacks.

- Finding a collision for the 128 bit shared secret visualization itself will not yield anything as the attacker is not in control of the SHA1 input. Instead, the collision has to come from the secret values chosen during the key exchange, and as such no known collision attack will help the adversary here. Succeeding with this however leads to a perfect man-in-the-middle attack, as we will describe in 6.2.3.

And so we see that there is no case where finding a regular SHA1 hash collision or pre-image can fool the protocol, either the found collision will contain too little information to be of use, or SHA1 is used in combination with another method that mitigates known attacks.

However, the way SHA1 is combined with decryption is insecure. We have already proven this with the attacks from sections 4.1.1 and 4.1.2, which exactly manage to find a collision C' for ciphertext C such that

$$\text{SHA1}(\text{Dec}_K(C)) = \text{SHA1}(\text{Dec}_K(C'))$$

In attempt to mitigate known theoretical attacks, the protocol has been made vulnerable to new attacks that trivially find collisions.

6.2 Known attacks on MTProto

Several attacks on MTProto have been proposed already, primarily with the intention of taking over a secret chat and being able to read all messages. The simplest approach to this is by performing a man-in-the-middle attack, that is establishing two secret chats with one unsuspecting user in each.

6.2.1 Malicious server MitM attack

An earlier version of MTProto used a modified version of the Diffie-Hellman key exchange in which an extra *nonce* was supplied by the server during the final key derivation step

$$key \leftarrow (\text{pow}(g_b, a) \bmod p) \oplus nonce$$

From a post on Russian blog habrahabr.ru [32], we saw that a malicious server could perform a man-in-the-middle attack and send out two different nonces so that the final keys end up identical, as seen in figure 6.2.

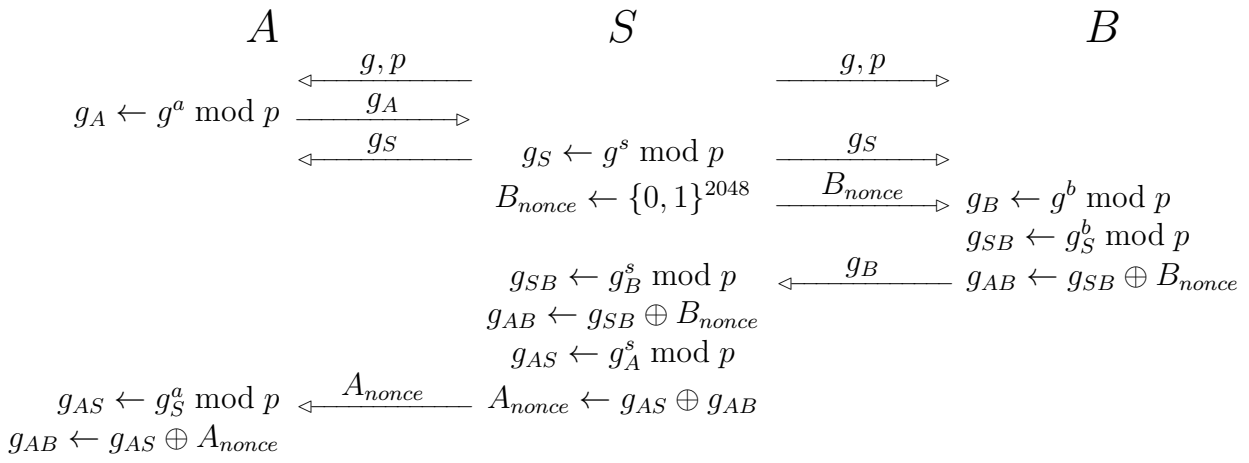

Figure 6.2: Malicious server MitM attack.

In the last step we see A compute the shared secret as $g_{AB} \leftarrow g_{AS} \oplus A_{nonce}$, and from the way A_{nonce} is chosen this is equivalent to

$$g_{AB} \leftarrow \cancel{g_{AS}} \oplus \cancel{g_{AS}} \oplus g_{AB}$$

The attack is only possible under the assumption that users will send their public DH value to the server before it gives them the nonce. This way the

server can trivially obtain the shared secret for a session, and gain access to all sent messages. This is a very serious flaw that Telegram claims to have introduced by mistake, and although it has since been removed from the protocol it has sparked suspicion that the authors of Telegram may have intentionally backdoored the protocol to always have a way of accessing end-to-end encrypted chats.

6.2.2 Naive third party MitM attack

A working man-in-the-middle attack has been demonstrated by Vico [30], as shown in figure 6.3, carried out on an unofficial Linux terminal client. It assumes the user would use a malicious client which sends all communication through a middle man, who ends up having full access to all messages.

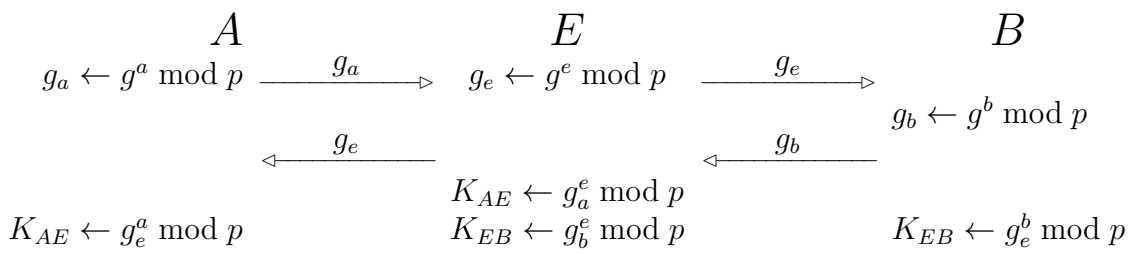

Figure 6.3: The naive MitM attack.

The attack will be detected if the users compare key fingerprints like the one seen in figure 3.2. This can be accomplished by the malicious client used by A receiving the key fingerprint of K_{EB} from the middle man, and displaying this instead of the one computed from K_{AE}. But in this case the malicious client might as well just forward all the decrypted messages to the middle man, and bypass security altogether.
This attack is mitigated by users not using a client from an untrusted source, and verifying key fingerprints as intended.

6.2.3 Undetected third party MitM attack

A second man-in-the-middle attack is outlined by Rizzo et al. [17]. Rather than assuming that users will not compare key fingerprints, their attack searches for fingerprint collisions such that the attack will be completely undetected, and shows how to manipulate the situation to have just a square-root of the complexity for collision finding.

The naive undetected approach is shown in figure 6.4. When user A initiates a secret chat with user B, and this request g_a is forwarded through a third user E, the best E can do is to send his own request g_{e_1} to B. When B responds with g_b, B and E have agreed on a key K_{EB}. E now has to come up with a reply g_{e_2} to A for which the key K_{AE} will have the same fingerprint as K_{EB}. This requires 2^{128} operations, or 2^{127} to have a 50% probability of finding a collision.

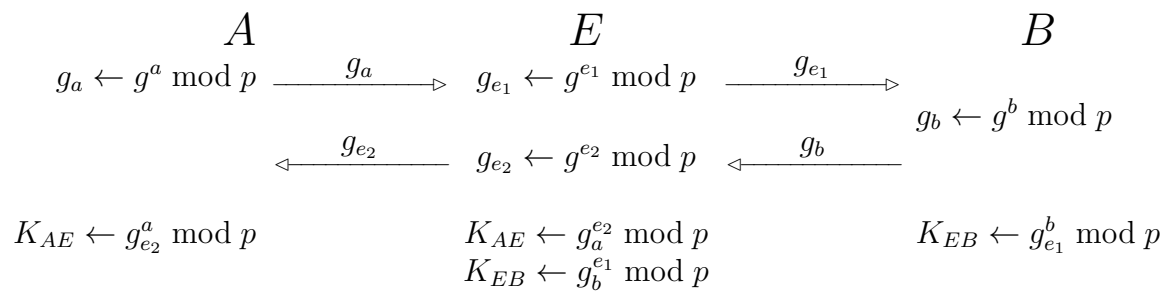

Figure 6.4: The naive undetected MitM attack.

The article assumes the adversary is capable of having A and B both initiate a secret chat at the same time, by means of social engineering. The significance of this is that now E hasn't committed to the value e_1 yet, and can freely choose both e_1 and e_2, as shown in figure 6.5. This allows for the adversary to do a birthday attack, reducing the complexity of finding an $auth\_key$ collision to its square-root, $\sqrt{2^{128}} = 2^{64}$. This needs to be done for both e_1 and e_2, and in total the cost of the attack is $2^{64}+2^{64} = 2^{65}$ operations to have a 50% probability of finding a collision.

We can also think of this as increasing the required security from the weaker *second pre-image resistance* notion to the stronger *collision resistance* notion, as described in section 2.4.3.

One might argue that this attack is ineffective if the adversary lacks computing power, as the two users will have to wait for him to find the collision before they can begin the conversation. Using the AntMiner S5 as we looked at previously, this would take months. But if say the whole Bitcoin mining community collaborated, this attack could be carried out mere seconds, ignoring the cost of the Diffie-Hellman computations.

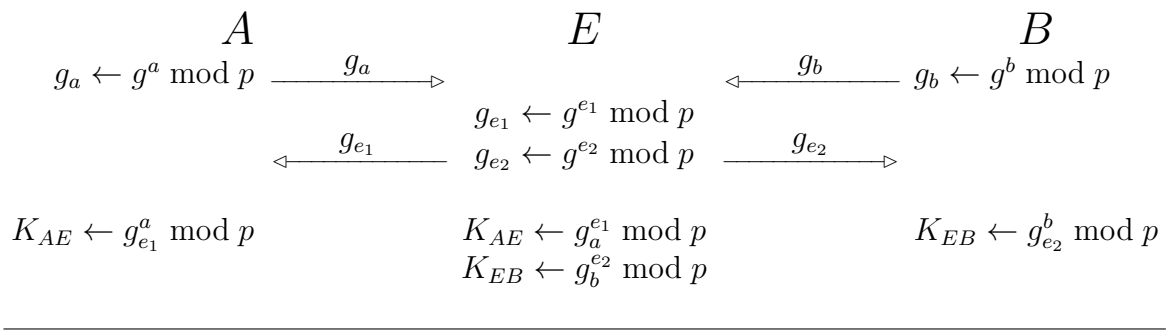

Figure 6.5: Having the freedom to choose e_1, e_2 simultaneously, only the square root of the number of operations is required.

Chapter 7

Proven crypto alternative

In this chapter we will look at another secure instant messaging application known as TextSecure, and see which primitives it uses and how it provides confidentiality, integrity, and authenticity. We will see how this compares to Telegram, and if Telegram could learn anything from TextSecure.

7.1 TextSecure

TextSecure (or Signal as the iOS flavor is named) is a secure messaging application for Android that uses state-of-the-art methods and makes no compromise in security over performance. Its developer WhisperSystems is even endorsed by Edward Snowden [10]! We will give a brief introduction to the primitives it uses:

Curve25519 TextSecure uses Curve25519, which is an elliptic curve, for Diffie-Hellman key exchange [6]. A user generates a random 256-bit secret key and given this, Curve25519 outputs a matching 256-bit public key. Given the secret key of one user and the public key of another Curve25519 outputs a 256-bit secret shared by the two users. We will refer to this last step of the elliptic curve Diffie-Hellman key exchange as $\mathrm{DH}(\cdot)$.

HMAC-SHA256 HMAC is a mac function that has the form

$$\mathrm{HMAC}(M) \leftarrow \mathrm{H}((K \oplus \mathit{opad}) \mid\mid \mathrm{H}((K \oplus \mathit{ipad}) \mid\mid M))$$

where H is any hash function with input block size n, K is a secret key padded with zeroes to get length n, *opad* is a string of the byte 0x36 repeated to get length n and *ipad* is the byte 0x5C repeated to get length n.

In this case the hash function used is SHA256, a variant of the successor to SHA1. This has input block size 512 bits and output size 256 bits, and unlike SHA1 it is not cryptographically broken. The construction is shown in figure 7.1, the output being *hash sum 2*.

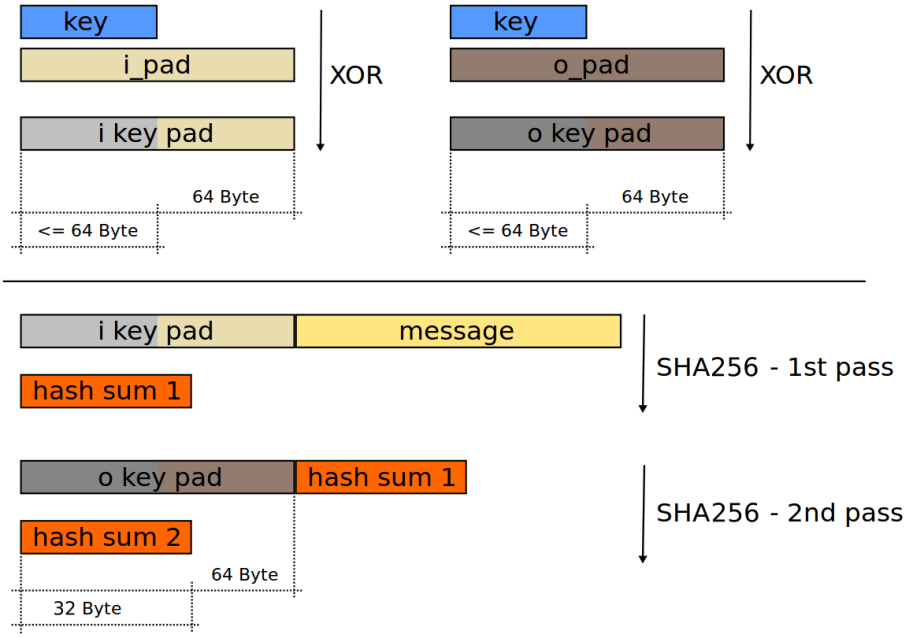

Figure 7.1: HMAC-SHA256 construction.

HKDF HKDF is an HMAC based key derivation function which outputs secure keys of arbitrary length. We will treat it as a black box and not go into more detail in this work, a thorough description can be found at the Internet Engineering Task Force page [11].
In TextSecure the hash function used in HKDF is SHA256 and the output size is 512 bits.

Axolotl Ratchet [16] is a Diffie-Hellman Ratchet, that is a DH based protocol for generating new master keys and providing forward secrecy, which can be done for every message sent and thus providing it on a per-message basis. In the Axolotl Ratchet protocol used in TextSecure, each user must maintain the following keys:

- ephemeralKey: one time key included in every message, used to derive the next rootKey.

- rootKey: the shared secret master key, used together with an ephemeral key from each user to derive new shared rootKey and chainKey.
- chainKey: key used to derive one or more messageKeys.
- messageKey: key used to derive encrypt/decrypt and MAC keys.

TextSecure only generates one messageKey per chainKey, and provides per-message forward secrecy.

7.1.1 Registration

A new user registers his phone number and a username to the TextSecure server. Along with this he generates a DH identity key pair and one or more (ephemeral) prekey pairs using Curve25519. He sends the public half of the identity and the prekeys to the server.

7.1.2 Key exchange

In the following section we will use the notation A_{id1} for user A's public half of his identity key, or A_{id0} for the secret half. Similarly, A_{pk1} will refer to the public half of one of user A's prekeys (ephemeral keys).

When one user wants to start messaging another, they perform a key exchange. User A wants to talk to user B. A is already registered to the server and asks the server for B's identity and one of his public prekeys.
The server sends A the public identity key of B, B_{id1}, and one of his public prekeys B_{pk1}. A now generates a new ephemeral key pair (A_{pk0}, A_{pk1}), and from these keys he can compute the shared secrets $rootKey$ and $chainKey$ using a triple DH:

$$3DH \leftarrow \mathrm{DH}(A_{id0}, B_{pk1}) \;||\; \mathrm{DH}(A_{pk0}, B_{id1}) \;||\; \mathrm{DH}(A_{pk0}, B_{pk1})$$

$$rootKey, chainKey \leftarrow \mathrm{HKDF}(3DH)$$

Where HKDF is the key derivation function we introduced earlier, which in this case is set to output 512 bits. The $rootKey$ is set to the first 256 bits of the output, the $chainKey$ is set to the next 256 bit.

Having computed this, A sends a message to B containing A_{id1}, A_{pk1}, an identifier of the prekey he used from B, and finally a message of the form described in section 7.1.4. B is now able to compute the same $rootKey$ and $chainKey$ using his private identity key, the corresponding private prekey

and the values received from A.

Furthermore, both the users now have the public identity key of one another and these are used to authenticate each other through an out-of-band channel that is hard to impersonate such as a phone call, or even better in person.

7.1.3 Key derivation

First a $messageKey$ is derived by computing HMAC-SHA256 on input byte 0x01 and keyed with $chainKey$:

$$messageKey \leftarrow \text{HMAC-SHA256}(chainKey, 0\text{x}01)$$

$messageKey$ is then used as input to HKDF to derive a 256-bit $encryptionKey$ and 256-bit $macKey$:

$$encryptionKey, macKey \leftarrow HKDF(messageKey, 0\text{x}0)$$

Whenever a message containing an ephemeral key is received, a new $rootKey$ and $chainKey$ is generated as part of the Axolotl Ratchet protocol. This is done by computing a shared secret HMAC key using the received ephemeral key and one of the user's own, and running HKDF on the existing $rootKey$ as input. This outputs a new $rootKey, chainKey$ pair.

7.1.4 Message encryption

Message payloads in TextSecure have the form:

- **prekey**: an ephemeral public key half used to derive the next shared $rootKey$.
- **counter**: 32-bit counter increasing for every message sent under the same ephemeralKey.
- **previousCounter**: max value of counter sent under the sender's last ephemeralKey.
- **ciphertext**: message encrypted with key derived from Axolotl Ratchet, using AES256-CTR with the high 32 bits of the nonce equal to the counter included in the payload.

This payload is then wrapped in the following construction:

- **version**: 8 bits, high 4 bits represent current message version, low 4 bits represent client max known protocol version.
- **message payload**: the payload that we saw above.
- **mac**: HMAC-SHA256 of the above fields using MAC key derived from Axolotl Ratchet, and truncated to 64 bits.

As we see, TextSecure uses the symmetric-key encryption scheme AES with 256-bit keys, just like Telegram. However, it uses AES in counter mode which does not have the BACPA insecurity like IGE, as it is not chained. This encryption provides confidentiality and IND-CPA security.

We also see that TextSecure uses a proper message authentication code, HMAC-SHA256. This MAC scheme is strongly unforgeable, and as it is computed from the ciphertext it provides integrity of ciphertexts.
And so we have that TextSecure provides both IND-CPA and INT-CTXT, and this combined means it has authenticated encryption.

7.1.5 Why is this better?

TextSecure is based on strong primitives that have withstood cryptanalysis from the crypto community for years, and these are combined in a way that provenly provides authenticated encryption.
Telegram on the other hand has crafted its own encryption scheme and deployed it in an unproven state, and prior to any scrutiny from other cryptographers. We have seen this done time and time again, and rarely with good results. Take for example the smart grid meters that were shown to use terrible crypto back in April this year [12].

Furthermore, the DH Ratchet is a very nice way of providing forward secrecy on a per-message basis with little overhead, which is an improvement over Telegram's one key per 100 messages approach.

7.2 Conclusion

In this work we have shown that Telegram, with its use of aging primitives, does not manage to provide data integrity of ciphertexts nor authenticated encryption, and is vulnerable to chosen-ciphertext attacks. The attempt to mitigate known attacks has introduced new vulnerabilities, and we suggest

that the Telegram team updates its protocol to use strong, modern primitives. For message authentication codes it should use a good HMAC, use a proper key derivation function, and update the key exchange to use elliptic curve Diffie-Hellman based on Curve25519. Telegram has a great emphasis on computational performance of its protocol, which is why CTR with its parallelization seems to be the logical choice of encryption mode. We suggest using CTR instead of IGE mode, as IGE offers no benefits over CTR.
Overall, we can conclude yet again that homegrown cryptography is a bad approach.

Bibliography

[1] Android Security Engineer Alex Klyubin. Some SecureRandom Thoughts. http://android-developers.blogspot.dk/2013/08/some-securerandom-thoughts.html, 2013. [Online; accessed 28-August-2015]. (page 35)

[2] Amazon. EC2: Previous Generation Instances. http://aws.amazon.com/ec2/previous-generation/, 2014. [Online; accessed 28-August-2015]. (page 62)

[3] Jean-Philippe Aumasson. BLAKE2 — fast secure hashing. https://blake2.net/, 2012. [Online; accessed 28-August-2015]. (page 62)

[4] Gregory V Bard. Modes of encryption secure against blockwise-adaptive chosen-plaintext attack. *IACR Cryptology ePrint Archive*, 2006:271, 2006. (page 17, 59)

[5] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology—ASIACRYPT 2000*, pages 531–545. Springer, 2000. (page 25, 28)

[6] D. J. Bernstein. A state-of-the-art Diffie-Hellman function. http://cr.yp.to/ecdh.html. [Online; accessed 3-September-2015]. (page 69)

[7] Blockchain. Bitcoin currency statistics. https://blockchain.info/stats, 2015. [Online; accessed 28-August-2015]. (page 63)

[8] Internet Engineering Task Force. PKCS #1: RSA Encryption. https://tools.ietf.org/html/rfc2313, 1998. [Online; accessed 10-September-2015]. (page 33)

[9] Electronic Frontier Foundation. Secure Messaging Scoreboard. https://www.eff.org/secure-messaging-scorecard, 2015. [Online; accessed 1-September-2015]. (page 5)

[10] Max Eddy from PCMag.com. Snowden to SXSW: Here's How To Keep The NSA Out Of Your Stuff. http://securitywatch.pcmag.com/security/321511-snowden-to-sxsw-here-s-how-to-keep-the-nsa-out-of-your-stuff, 2014. [Online; accessed 11-September-2015]. (page 69)

[11] Internet Engineering Task Force (IETF). HMAC-based Extract-and-Expand Key Derivation Function (HKDF). http://tools.ietf.org/html/rfc5869, 2010. [Online; accessed 7-September-2015]. (page 70)

[12] Philipp Jovanovic and Samuel Neves. Dumb crypto in smart grids: Practical cryptanalysis of the open smart grid protocol. Cryptology ePrint Archive, Report 2015/428, 2015. http://eprint.iacr.org/. (page 73)

[13] J. Katz and Y. Lindell. *Introduction to Modern Cryptography: Principles and Protocols*. Chapman & Hall/CRC Cryptography and Network Security Series. Taylor & Francis, 2007. ISBN 9781584885511. URL https://books.google.dk/books?id=TTtVKHdOcDoC. (page 9, 15, 19, 24, 26, 27, 28, 29)

[14] Ben Laurie. Openssl's implementation of infinite garble extension. 2006. (page 12)

[15] Kenny Paterson. Introduction: secure channels, generic composition, basic attacks. lecture from bar-ilan university, 2014. URL http://crypto.biu.ac.il/4th-biu-winter-school. (page 13, 23)

[16] Trevor Perrin and Moxie Marlinspike. Axolotl Ratchet. https://github.com/trevp/axolotl/wiki, 2013. [Online; accessed 7-September-2015]. (page 70)

[17] Juliano Rizzo and Alexander Jung-Loddenkemper. A 2^{64} Attack On Telegram, And Why A Super Villain Doesn't Need It To Read Your Telegram Chats. http://bit.ly/1wQspqc, 2015. [Online; accessed 28-August-2015]. (page 66)

[18] John J. G. Savard. A Cryptographic Compendium. http://friedo.szm.com/krypto/JS/co0409.htm, 2000. [Online; accessed 28-August-2015]. (page 11)

[19] Bruce Schneier. When Will We See Collisions for SHA-1? https://www.schneier.com/blog/archives/2012/10/when_will_we_se.html, 2012. [Online; accessed 28-August-2015]. (page 62)

[20] Statistica. Average retail electricity prices in the U.S. from 1990 to 2014. http://www.statista.com/statistics/183700/us-average-retail-electricity-price-since-1990/, 2015. [Online; accessed 28-August-2015]. (page 63)

[21] Marc Stevens. New collision attacks on sha-1 based on optimal joint local-collision analysis. In *Advances in Cryptology–EUROCRYPT 2013*, pages 245–261. Springer, 2013. (page 62)

[22] Telegram team. Telegram Android source code. https://github.com/DrKLO/Telegram, 2015. [Online; accessed 14-September-2015]. (page 32)

[23] TechCrunch. Telegram Says It's Hit 62M MAUs And Messaging Activity Has Doubled. http://techcrunch.com/2015/05/13/telegram-says-its-hit-62-maus-and-messaging-activity-has-doubled/, 2015. [Online; accessed 1-September-2015]. (page 5)

[24] Telegram. Secret chats, end-to-end encryption. https://core.telegram.org/api/end-to-end, 2013. [Online; accessed 28-August-2015]. (page 34)

[25] Telegram. Perfect Forward Secrecy. https://core.telegram.org/api/end-to-end/pfs, 2013. [Online; accessed 28-August-2015]. (page 44)

[26] Telegram. FAQ for the Technically Inclined. https://core.telegram.org/techfaq, 2013. [Online; accessed 28-August-2015]. (page 62, 64)

[27] Telegram. Mobile Protocol: Detailed Description. https://core.telegram.org/mtproto/description, 2013. [Online; accessed 28-August-2015]. (page 43, 47)

[28] Telegram. Sequence numbers in Secret Chats. https://core.telegram.org/api/end-to-end/seq_no, 2013. [Online; accessed 28-August-2015]. (page 40)

[29] Telegram. 10 Billion Telegrams Delivered Daily. https://telegram.org/blog/10-billion, 2015. [Online; accessed 1-September-2015]. (page 5)

[30] Jesus Diaz Vico. Telegram: bypassing the authentication protocol. 2014. (page 66)

[31] Bitcoin wiki. Mining hardware comparison. https://en.bitcoin.it/wiki/Mining_hardware_comparison, 2015. [Online; accessed 28-August-2015]. (page 63)

[32] User x7mz. Is Telegram secure? (in Russian). http://habrahabr.ru/post/206900/, 2013. [Online; accessed 2-September-2015]. (page 65)