SCIS 2017 2017 Symposium on Cryptography and Information Security
Naha, Japan, Jan. 24 - 27, 2017
The Institute of Electronics, Information and Communication Engineers

Security Analysis of End-to-End Encryption in Telegram

Jeeun Lee * Rakyong Choi * Sungsook Kim * Kwangjo Kim *

Abstract: Telegram is known as one of the most popular instant messaging (IM) services for secure communications. It features end-to-end encryption (E2EE) in secret chats based on their customised protocol called MTProto. This brand new protocol is believed as a safe alternative among the public, however, it is in doubt and has not been fully reviewed by cryptanalytic experts. It is theoretically demonstrated in 2015 that MTProto does not meet indistinguishability under chosen ciphertext attack (IND-CCA) and integrity of ciphertexts (INT-CTXT). In this paper, we start to analyse the security vulnerabilities of E2EE in simplified Telegram in a heuristic manner and suggest typical building blocks for E2EE in general IM system.

Keywords: MTProto, IND-CCA, INT-CTXT, authenticated encryption

1 Introduction

As smartphones came into widespread use in the late 2000s, a number of instant messaging (IM) services such as WhatsApp, KakaoTalk, LINE, Facebook Messenger, and Telegram have burst onto mobile app stores. The various IM clients can be classified into three types according to their provided encryption protocols: no encryption, client-to-server encryption, and client-to-client or end-to-end encryption (E2EE). Since the lack of privacy protection has been issued constantly, now the majority of IM services provide E2EE based on verified cryptographic protocols. Telegram is particularly regarded as one of the most secure services in public and has over 100 million active users. Based on Telegram's customised protocol called MTProto, it provides client-to-server encryption in cloud chats for syncing all connected devices and E2EE in secret chats for only two devices that used to initiate or accept the secret chat. This brand new protocol, however, is actually in doubt and has not been fully scrutinised by cryptanalytic experts yet. Given that there already exist other protocols that thoroughly audited and universally praised as secure, avoiding criticism for MTProto seems unlikely unless extensive investigation is done.

One of the most popular cryptographic protocols is Signal Protocol (formerly known as the Axolotl Protocol) developed by Open Whisper Systems in 2013 [1] and currently implemented into Signal, WhatsApp, Google Allo, and Facebook Messenger. As of October 2016, its latest version is considered as sound and has no major flaws according to the researchers from three different universities [2]. Meanwhile, Telegram's MTProto has been criticised until now and Jakobsen *et al.* theoretically demonstrated Telegram 2.7.0 (visited GitHub in April 2015) is not indistinguishability under chosen-ciphertext attack (IND-CCA) and integrity of ciphertexts (INT-CTXT) secure [3, 4]. From the fact that MTProto does not check neither the length nor the content of the padding during block cipher decryption, two attacks were tried: (a) adding a random block at the end of the ciphertext and (b) replacing the last block with a random block. The first weakness can be fixed easily by adding the process to check the length of the padding during decryption and discard the message when it is longer than expected. As for mitigating the second weakness, the encryption process should be changed, which makes communications between patched and unpatched clients difficult. Thus, it is desirable to replace the current scheme with the entirely different, better one that guarantees authenticated encryption (AE).

In this paper, we mainly focus on MTProto's vulnerabilities in terms of IND-CCA and INT-CTXT. In Section 2, the notion of special mode of operation and AE is provided. Section 3 describes the overall process of MTProto and Section 4 introduces two known attacks from the view of IND-CCA and INT-CTXT. Then, the implementation of simplified MTProto and typical building blocks for E2EE in IM system are explained in Sections 5 and 6, respectively. Finally, concluding remarks will be made in Section 7.

2 Preliminaries

2.1 Modes of Operation: IGE mode

Block ciphers like DES and AES are one of the important cryptographic primitives and widely used to encrypt bulk data. Since their operations only work on a fixed-length group of bits called a block, there can be many approaches to combine repeated operations for multiple blocks (*i.e.*, modes of operation). As a simple example, Electronic Codebook (ECB) mode divides the message into blocks and encrypts each of them indepen-

* School of Computing, KAIST, Daejeon 34141, South Korea. {jeeun.lee,thepride,kusino,kkj}@kaist.ac.kr

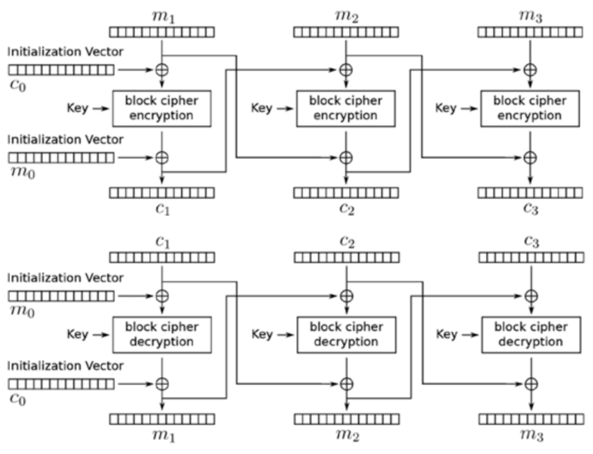


Figure 1: Encryption and decryption in IGE mode

dently, hence the same ciphertext blocks are generated from the same plaintext blocks. This property makes the system highly vulnerable and insecure against traffic analysis. Of several modes of operation more secure than ECB mode, Infinite Garble Extension (IGE) mode is used in Telegram's MTProto.

IGE mode was initially introduced by Campbell in 1978 to prevent spoofing attacks [5]. It has the property that errors are propagated forward indefinitey, that is, any difference in ciphertext changes (*i.e.*, garbles) the decryption of all subsequent ciphertext [6]. The blocks are chained as follows:

$$c_i \leftarrow f_K(m_i \oplus c_{i-1}) \oplus m_{i-1}$$

for encryption and

$$m_i \leftarrow f_K^{-1}(c_i \oplus m_{i-1}) \oplus c_{i-1}$$

for decryption as in Figure 1 [3] where f_K and f_K^{-1} are block cipher encryption and decryption function with key K, respectively. From this equation, the first output block c_1 needs two non-existent inputs, c_0 and m_0. The initialisation vector (IV) is defined using a second random key K_0: $c_0 = f_{K_0}(m_0)$ or arbitrary given parameters.

After over 30 years now IGE mode is rarely used and replaced with other AE modes because recovering message with carefully chosen errors is possible.

2.2 Authenticated Encryption

Although the previous modes of operation give confidentiality for block ciphers, much better modes that simultaneously provide confidentiality, integrity, and authenticity known as AE were developed. In 2000, Bellare and Namprempre introduced the notion of AE to guarantee both confidentiality of the message and integrity of the sender while transmission over an insecure channel like mobile network [7]. As a very natural way to construct AE, they suggested a generic composition paradigm on secure encryption and secure MAC protocols such as AES and HMAC. They used indistinguishability under chosen-plaintext attack (IND-CPA),

Table 1: Security results for the composite AE schemes

	IND -CPA	IND -CCA	NM -CPA	INT -PTXT	INT -CTXT
E&M	insecure	insecure	insecure	secure	insecure
MtE	secure	insecure	insecure	secure	insecure
EtM	secure	secure	secure	secure	secure

non-malleability under chosen-plaintext attack (NM-CPA), or indistinguishability under chosen-ciphertext attack (IND-CCA) for confidentiality like the classical block ciphers as security requirements of AE, then introduced two notions for integrity, namely integrity of plaintexts (INT-PTXT) and integrity of ciphertexts (INT-CTXT) assuming that the adversary $\mathcal{A}$ is allowed a chosen-message attack as below:

Definition 1 (INT-PTXT)**.** AE satisfies INT-PTXT security if the advantage of any probabilistic polynomial-time adversary $\mathcal{A}$ to produce a ciphertext $c = \mathcal{E}(m)$, where m is not previously produced by the sender, is negligible.

Definition 2 (INT-CTXT)**.** AE satisfies INT-CTXT security if the advantage of any probabilistic polynomial-time adversary $\mathcal{A}$ to produce a ciphertext $c = \mathcal{E}(m)$ not previously produced by the sender is negligible, regardless of whether the underlying plaintext m is new or not.

From the above security requirements, they designed and analysed three composition methods on encryption and MAC protocols, namely Encrypt-and-MAC (E&M), MAC-then-Encrypt (MtE), and Encrypt-then-MAC (EtM) as below:

E&M: For encryption with authentication, we encrypt a plaintext m as $Enc(m)$ where Enc is an encryption algorithm of secure encryption protocol and append a tag t of m using MAC, *i.e.*, a ciphertext $\mathcal{E}(m) = Enc(m)\|t$. For decryption with verification, we check the validity of the tag as well as the decryption of the ciphertext.

MtE: For encryption with authentication, we encrypt a plaintext m as $Enc(m)$ and append a tag t of $Enc(m)$ instead of m, *i.e.*, a ciphertext $\mathcal{E}(m) = Enc(m)\|t_{enc}$ where t_{enc} is a tag of $Enc(m)$. For decryption with verification, we first verify the tag and then decrypt the ciphertext.

EtM: For encryption with authentication, we append a tag t of m first, then encrypt an appended plaintext $m\|t$, *i.e.*, a ciphertext $\mathcal{E}(m) = Enc(m\|t)$. For decryption with verification, we first decrypt the ciphertext to get the plaintext and the tag.

Table 1 [7] describes security results for the composite AE schemes when the given MAC is assumed to be strongly unforgeable and shows that EtM can only reach the highest definition of security in AE.

Aside from this research, Bellare & Rogaway and An & Bellare suggested ‘encode-then-encipher’ [8] and ‘encryption with redundancy’ [9] approaches, respectively, but both of them are rather insecure and inefficient than the general composition paradigm [10]. Thus, we focus on a generic composition paradigm for the rest of the paper.

3 MTProto in Secret Chats

MTProto has two different encryption proots for cloud chats and secret chats. Cloud chats use client-to-server encryption for syncing all connected devices and secret chats use E2EE for only two devices that used to initiate or accept the secret chat. Throughout this paper, we mainly focus on MTProto in Secret Chats using E2EE denoted by ‘MTProto’ simply.

MTProto uses Diffie-Hellman (DH) key exchange, Secure Hash Algorithm 1 (SHA-1), Key Derivation Function (KDF), and AES-256 in IGE mode as cryptographic primitives and the overall process is described in Figure 2 [11].

3.1 Key Generation

The DH key exchange is used for generating an ephemeral key. After key exchange, the sender and the receiver share the same 2048-bit symmetric key K. In order to protect past communications, secret key is regenerated once a key has been used for more than 100 messages or more than a week.

The payload x is generated by concatenating some auxiliary information, random bytes, message, and padding such that $|x| \bmod B = 0$ where B is the block length. Then the payload except padding is computed by hash function SHA-1 whose output named *tag*.

This *tag* is hashed again by KDF for generating AES key and IV. The input of KDF is (K, tag) and the output is (k, c_0, m_0) of the length (κ, B, B) where $\kappa = 256$ bits and $B = 128$ bits.

3.2 Encryption

The AES-256 in IGE mode is used for encryption. Let $x_1, ..., x_l$ be the l blocks of the payload, each of length B, then ciphertext is computed as below:

$$c_i \leftarrow F_k(m_i \oplus c_{i-1}) \oplus m_{i-1}$$

where F is a pseudorandom permutation, *e.g.*, AES.

The final output of the encryption is c including other information.

$$c = (tag, c_1, ..., c_l)$$

3.3 Decryption

Given ciphertext c, *tag* is used again in KDF. Using (tag, K), KDF output is (k, c_0, m_0) same as encryption. Also, the IGE mode is used again for decryption and the payload x is recovered.

$$m_i \leftarrow F_k^{-1}(c_i \oplus m_{i-1}) \oplus c_{i-1}$$

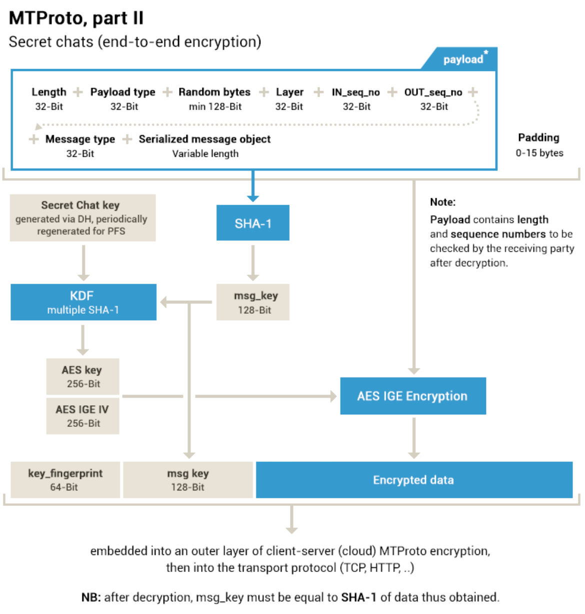

MTProto, part II
Secret chats (end-to-end encryption)
payload*
Length 32-Bit
Payload type 32-Bit
Random bytes min 128-Bit
Layer 32-Bit
IN_seq_no 32-Bit
OUT_seq_no 32-Bit
Message type 32-Bit
Serialized message object Variable length
Padding 0-15 bytes
Secret Chat key generated via DH, periodically regenerated for PFS
SHA-1
Note: Payload contains length and sequence numbers to be checked by the receiving party after decryption.
KDF multiple SHA-1
msg_key 128-Bit
AES key 256-Bit
AES IGE IV 256-Bit
AES IGE Encryption
key_fingerprint 64-Bit
msg key 128-Bit
Encrypted data
embedded into an outer layer of client-server (cloud) MTProto encryption, then into the transport protocol (TCP, HTTP, ..)
NB: after decryption, msg_key must be equal to SHA-1 of data thus obtained.

Figure 2: MTProto in E2EE

The payload except padding is computed by hash function and checks whether the result is same as *tag* in ciphertext c. If so, we can verify that the message in payload is original plaintext.

4 Known Attacks on MTProto

Jakobsen *et al.* theoretically demonstrated that MTProto does not meet IND-CCA and INT-CTXT in 2015 [3, 4]. Based on random padding vulnerabilities that MTProto does not check neither the length nor the content of the padding during AES-256 in IGE mode decryption, two attacks were tried: padding length extension and last block substitution.

4.1 Attack 1: Padding Length Extension

From **Definition 2**, they created a new ciphertext to break INT-CTXT security of MTProto. In order to understand INT-CTXT security, let us restate why MTProto is not IND-CCA secure using **Lemma 1**.

Lemma 1. *For a probabilistic polynomial-time adversary $\mathcal{A}$, $\mathcal{A}$ always wins the following game, i.e., MTProto is not IND-CCA secure under the following game.*

1. $\mathcal{A}$ outputs different messages M_0 and M_1 of the same length.
2. The challenger $\mathcal{C}$ chooses $b \in \{0, 1\}$ randomly and outputs the ciphertext $C_b \leftarrow \mathcal{E}(M_b)$.
3. $\mathcal{A}$ appends a 128-bit random block c_r to C_b and ask $\mathcal{C}$ to decrypt $C' = C_b \| c_r$.
4. $\mathcal{C}$ returns M' where $M' = M_b$ for any b.
5. $\mathcal{A}$ guesses b as 0 if $M' = M_0$, 1 otherwise.

This attack is possible since extra padding on ciphertext yields only the extension of the padding of plaintext without changing the plaintext. Obviously, $\mathcal{A}$ gets a ciphertext $C' = \mathcal{E}(M_b)$ for M_b and this ciphertext has not been previously produced by the sender because of the random padding.

Corollary 1. *MTProto protocol is not INT-CTXT secure from* ***Lemma 1****.*

To be secure against this attack, it is necessary to check the length of padding in M' by modifying the decryption process. Decryption algorithm will discard the message if the length of padding is larger than the block size.

4.2 Attack 2: Last Block Substitution

Since the padding is not authenticated in the process of MTProto, it is possible to make a collision with a non-negligible probability as **Lemma 2**, by modifying the last 128-bit (16-byte) blocks.

Lemma 2. *For a probabilistic polynomial-time adversary $\mathcal{A}$, $\mathcal{A}$ wins the following game with a probability at most 2^{-8}, i.e., MTProto is not INT-CTXT secure under the following game.*

1. $\mathcal{A}$ outputs a message M whose length in bytes is equal to b mod 16.
2. The challenger $\mathcal{C}$ hashes M into the message key $msg\_key \leftarrow$ SHA-1(M) to provide integrity of the plaintext.
3. Before encryption, $16-b$ random bytes of padding r are added to M, then sends $C = \mathcal{E}(M\|r)$.
4. $\mathcal{A}$ modifies last 16-byte blocks of C to get $C' \neq C$.
5. $\mathcal{A}$ outputs C'.

Proof. From the above game, $\mathcal{C}$ decrypts C' as $M'\|r'$. Then, only the last byte of M' is different from M by the non-malleability of IGE mode.

Thus, they claim that it is possible to have $M' = M$ with the probability at most 2^{-8} when $\mathcal{A}$ chooses a message M with the length in bytes equal to 1 mod 16, *i.e.*, they can generate a valid ciphertext $C' \neq C$ with the probability at most 2^{-8}. □

To make the protocol secure against this attack, it is sufficient to add padding to the computation of the authentication tag. But, since this requires to change the whole encryption with authentication process, it becomes impossible to communicate with the older versions of the protocol due to version compatibility.

5 Implementation

For experimental demonstration of random padding vulnerabilities introduced in Section 4, we simplify MTProto as a first step while preserving the major components, DH key exchange, SHA-1 of payload, customised KDF, and encrypted communication through the network with AES-256 in IGE mode. Compared to the original protocol, there are two insignificant differences: we (a) use RFC-3526 primes and generators in DH key exchange [12] and do not generate them everytime, and secondly, (b) consider the payload as only the length and the content of message. These are trivial and negligible when testing random padding vulnerabilities.

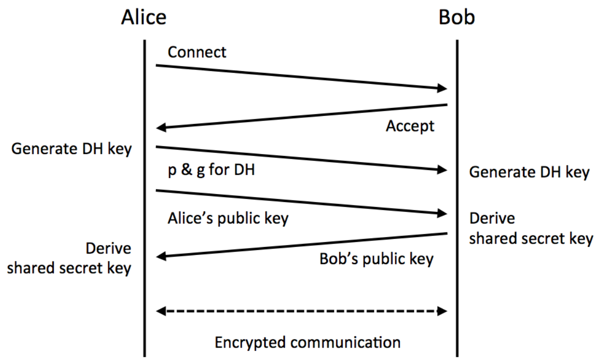


Figure 3: Procedure of key generation and exchange

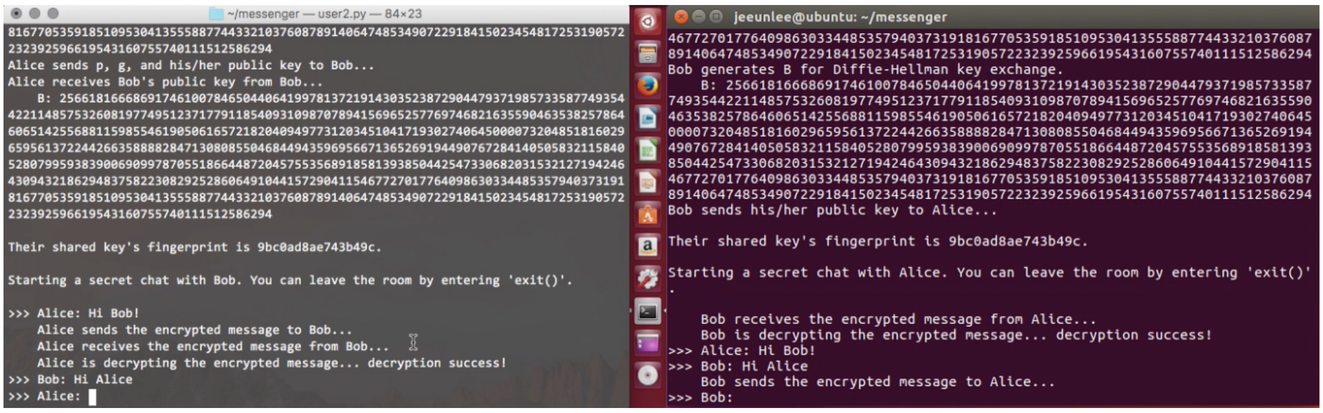
Simplified MTProto was developed in macOS Sierra and Ubuntu 16.10 using Python 3.5.2. It is designed to mock the communication between Alice and Bob through the Internet. In our experimental setup, the communication was simulated on a private network between host and virtual machine.

5.1 Key Generation

Figure 3 describes the procedure of key generation and exchange. Bob acts as a server and listens for incoming connections from the network. When Alice wants to talk to Bob, she initiates a Transmission Control Protocol (TCP) connection to Bob and sends her DH public key. Then Bob accepts Alice's connection and receives her public key. Bob also generates his own DH key and sends the public key back to Alice. Since Alice and Bob both know each other's public key, they can now derive a shared secret key and start encrypted communication. After they started a secret chat, one sends a message and generates a message key from SHA-1 of payload. The AES key and IV for AES-256 in IGE mode are derived by KDF using DH key and message key, and the pseudocode of KDF is shown in Algorithm 1 where the chraracter `+` denotes the string concatenation, `msg_key` is the last 16 bytes of SHA-1 of the payload and `dh_key` is the shared secret key between Alice and Bob. This encryption key varies message by message in MTProto, as well as in our implementation.

Algorithm 1: Pseudocode of KDF

```
def kdf(dh_key, msg_key):
  a = sha1(msg_key + dh_key[0:32])
  b = sha1(dh_key[32:48] + msg_key + dh_key[48:64])
  c = sha1(dh_key[64:96] + msg_key)
  d = sha1(msg_key + dh_key[96:128])
  aes_key = a[0:8] + b[8:20] + c[4:16]
  aes_iv  = a[8:20] + b[0:8] + c[16:20] + d[0:8]
  return (aes_key, aes_iv)
```



```
8167705359185109530413555887744332103760878914064748534907229184150234548172531905722
232392596619543160755740111512586294
Alice sends p, g, and his/her public key to Bob...
Alice receives Bob's public key from Bob...
    B: 25661816668691746100784650440641997813721914303523872904479371985733587749354
422114857532608197749512371779118540931098707894156965257769746821635590463538257864
606514255688115985546190506165721820409497731203451041719302740645000073204851816029
659561372244266358888284713080855046844943596956671365269194490767284140505832115840
528079959383900690997870551866448720457553568918581393850442547330682031532127194246
430943218629483758223082925286064910441572904115467727017764098630334485357940373191
816770535918510953041355588774433210376087891406474853490722918415023454817253190572
232392596619543160755740111512586294

Their shared key's fingerprint is 9bc0ad8ae743b49c.

Starting a secret chat with Bob. You can leave the room by entering 'exit()'.

>>> Alice: Hi Bob!
    Alice sends the encrypted message to Bob...
    Alice receives the encrypted message from Bob...
    Alice is decrypting the encrypted message... decryption success!
>>> Bob: Hi Alice
>>> Alice: █
```

```
467727017764098630334485357940373191816770535918510953041355588774433210376087
891406474853490722918415023454817253190572232392596619543160755740111512586294
Bob generates B for Diffie-Hellman key exchange.
    B: 25661816668691746100784650440641997813721914303523872904479371985733587
749354422114857532608197749512371779118540931098707894156965257769746821635590
463538257864606514255688115985546190506165721820409497731203451041719302740645
000073204851816029659561372244266358888284713080855046844943596956671365269194
490767284140505832115840528079959383900690997870551866448720457553568918581393
850442547330682031532127194246430943218629483758223082925286064910441572904115
467727017764098630334485357940373191816770535918510953041355588774433210376087
891406474853490722918415023454817253190572232392596619543160755740111512586294
Bob sends his/her public key to Alice...

Their shared key's fingerprint is 9bc0ad8ae743b49c.

Starting a secret chat with Alice. You can leave the room by entering 'exit()'
.

    Bob receives the encrypted message from Alice...
    Bob is decrypting the encrypted message... decryption success!
>>> Alice: Hi Bob!
>>> Bob: Hi Alice
    Bob sends the encrypted message to Alice...
>>> Bob:
```

Figure 4: Screenshot of a simplified MTProto execution

5.2 Message Encryption and Decryption

When Alice or Bob writes a message, MTProto attaches some information about the message at first and calls it as a payload. For simplicity, we put only 32-bit field of a message length to the payload. Then the padded payload is encrypted using AES-256 in IGE mode.

After obtaining the key and IV, we append a few bytes of random padding to the end of the payload to make it align with 16-byte blocks. For the encryption and decryption, we use `aes_ige_enc()` and `aes_ige_dec()` as shown in Algorithm 2. Since AES in IGE mode is not used much in practice, it is not even included in popular cryptographic libraries. We implemented the IGE mode by manipulating the input and output of AES in ECB mode using PyCrypto 2.6.1 library [13] in Python. `key` and `iv` are the values derived from `kdf()` and `M` is the aligned payload with padding.

Algorithm 2: Pseudocode of AES-256 in IGE mode

```
def aes_ige_enc(key, iv, M):
  aes = AES.new(key)  # ECB mode
  c_prev = iv[0:16]
  m_prev = iv[16:32]
  C = bytes()
  for i in range(0, len(M), 16):
    m = M[i:i+16]
    aes_ecb_enc_in = xor(m, c_prev)
    aes_ecb_enc_out = aes.encrypt(aes_ecb_enc_in)
    c = xor(aes_ecb_enc_out, m_prev)
    m_prev = m
    c_prev = c
    C += c
  return C

def aes_ige_dec(key, iv, C):
  aes = AES.new(key)  # ECB mode
  c_prev = iv[0:16]
  m_prev = iv[16:32]
  M = bytes()
  for i in range(0, len(C), 16):
    c = C[i:i+16]
    aes_ecb_dec_in = xor(c, m_prev)
    aes_ecb_dec_out = aes.decrypt(aes_ecb_dec_in)
    m = xor(aes_ecb_dec_out, c_prev)
    m_prev = m
    c_prev = c
    M += m
  return M
```

After the encrypted bytes of padded payload is returned by `aes_ige_enc()`, the data sent through the network include the fingerprint for DH key, `msg_key` and the encrypted payload. When Bob or Alice receives the data, the message is decrypted in reverse order. The fingerprint is verified first, and the key and IV for AES are derived by KDF using `msg_key` and shared secret key. Then finally the message is decrypted by `aes_ige_dec()`.

The screenshot of a simplified MTProto execution is shown in Figure 4. Our implementation can be also extended to have a dedicated server for relaying messages between users as Telegram cloud servers do. In this case, we can demonstrate an attacker in the server who can see all exchanges between Alice and Bob. While the communication is encrypted, we can possibly test the known attacks or find new exploit using the implementation.

6 Discussion

As an example to construct building blocks for E2EE in IM system, we suggest a typical construction shown in Table 2. These four building blocks are (a) Math Layer, (b) Cryptographic Primitive Layer, (c) Key Management Layer, and (d) Security Service Layer. Math Layer contains basic arithmetic for cryptographic primitives. Cryptographic Primitive Layer executes popular cryptographic primitives for secure E2EE. Key Management Layer is to execute key generation, distribution, and agreement including any key-related operations. Finally, Security Service Layer provides required security features for specific IM.

This will be used to handle additional attacks or measure cryptographic strength easily while maintaining the core design of cryptographic protocols.

7 Concluding Remarks

Although Telegram is particularly regarded as one of the most secure IM services in public, their own protocol called MTProto has not been fully reviewed by cryptanalytic experts. There are many questionable choices in the cryptographic protocol design, for example, SHA-1 whose collisions already found in 2005 [14], customised KDF, non-standard padding algorithm, and IGE mode which does not provide authenticity.

Table 2: Building blocks for E2EE in IM

Security Service Layer	Authenticated encryption Forward secrecy Backward compatibility MAC, HMAC Multiple encryption Modes of operation, *etc.*
Key Management Layer	Key generation Key distribution Key agreement KDF, *etc.*
Cryptographic Primitive Layer	Symmetric: AES Asymmetric: RSA, ElGamal, ECC Hash: SHA-1, SHA-3, *etc.*
Math Layer	Multiple precision arithmetic Finite field arithmetic Modular exponentiation Elliptic curve arithmetic, *etc.*

Among them, we focused on random padding vulnerabilities and known attacks, padding length extension and last block substitution related to AE. Since this weaknesses were demonstrated theoretically in 2015, we tried to simulate attacks through encrypted communications between Alice and Bob. It is required to construct building blocks for E2EE in general IM system so that additional attacks or cryptographic strength measure can be handled easily while maintaining the core design of cryptographic protocols. As a first step, simplified MTProto was developed for further tests of random padding vulnerabilities.

Designing new cryptographic protocols is very hard and it generally takes a long time to assure their reliability. Telegram asserts that they made MTProto in secure way in spite of some weaknesses of cryptographic primitives. Even its algorithm designed well and looks fine, it might still have flaws so we should be more careful. Given that there already exists other protocols universally praised as secure, MTProto should be extensively investigated to avoid criticism.

Acknowledgement

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIP) (No. NRF-2015R1A2A2A01006812).

References

[1] "Signal Protocol." `https://whispersystems.org`.

[2] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila, "A Formal Security Analysis of the Signal Messaging Protocol," in *IACR Cryptology ePrint Archive*, 2016/1013.

[3] J. Jakobsen, "A practical cryptanalysis of the Telegram messaging protocol," Master's thesis, Aarhus University, 2015.

[4] J. Jakobsen and C. Orlandi, "On the CCA (in)security of MTProto," in *IACR Cryptology ePrint Archive*, 2015/1177.

[5] C. Campbell, "Design and specification of cryptographic capabilities," *IEEE Communications Society Magazine*, vol. 16, pp. 15–19, November 1978.

[6] V. D. Gligor, P. Donescu, and J. Katz, "On message integrity in symmetric encryption," in *1st NIST Workshop on AES Modes of Operation*, Baltimore, Maryland, US, 2000.

[7] M. Bellare and C. Namprempre, "Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm," in *ASIACRYPT*, vol. 1976 of *Lecture Notes in Computer Science*, pp. 531–545, Springer, 2000.

[8] M. Bellare and P. Rogaway, "Encode-Then-Encipher Encryption: How to Exploit Nonces or Redundancy in Plaintexts for Efficient Cryptography," in *ASIACRYPT*, vol. 1976 of *Lecture Notes in Computer Science*, pp. 317–330, Springer, 2000.

[9] J. H. An and M. Bellare, "Does Encryption with Redundancy Provide Authenticity?," in *EUROCRYPT*, vol. 2045 of *Lecture Notes in Computer Science*, pp. 512–528, Springer, 2001.

[10] H. J. Kim and K. Kim, "Who can survive in CAESAR competition at round-zero?," in *the 31th Symposium on Cryptography and Information Security (SCIS)*, Kagoshima, Japan, 2014.

[11] "Secret chats, end-to-end encryption." `https://core.telegram.org/api/end-to-end`.

[12] "RFC-3526 MODP Diffie-Hellman groups for IKE." `https://tools.ietf.org/html/rfc3526`.

[13] "Python PyCrypto package." `http://pythonhosted.org/pycrypto`.

[14] X. Wang, Y. L. Yin, and H. Yu, "Finding Collisions in the Full SHA-1," in *CRYPTO*, vol. 3621 of *Lecture Notes in Computer Science*, pp. 17–36, Springer, 2005.