Automated Symbolic Verification of Telegram's MTProto 2.0

Marino Miculan

Department of Mathematics, Computer Science and Physics, University of Udine, Italy
marino.miculan@uniud.it

Nicola Vitacolonna

Department of Mathematics, Computer Science and Physics, University of Udine, Italy
nicola.vitacolonna@uniud.it

Abstract

MTProto 2.0 is a suite of cryptographic protocols for instant messaging at the core of the popular Telegram messenger application, which is currently used by more than 400 millions of people.

In this paper we analyse MTProto 2.0 using ProVerif, a symbolic cryptographic protocol verifier based on the Dolev-Yao model. In particular, we provide a fully automated proof of the soundness of MTProto 2.0's authentication, normal chat, end-to-end encrypted chat, and re-keying mechanisms with respect to several security properties, including authentication, integrity, confidentiality and perfect forward secrecy. To prove these results we proceed in a modular way: each protocol is examined in isolation, relying only on the guarantees provided by the previous ones and the robustness of the basic cryptographic primitives.

Our research proves the formal correctness of MTProto 2.0, and it can serve as a reference for implementation and analysis of clients and servers. Moreover, we isolate the aspects of cryptographic primitives that require further investigation, in order to deem this protocol suite definitely secure.

2012 ACM Subject Classification Security and privacy → Security protocols; Security and privacy → Logic and verification; Security and privacy → Formal security models

Keywords and phrases Specification, Verification and Synthesis; Security protocols; Practical verification; Privacy; Formal methods.

1 Introduction

Telegram [20] is a very popular instant messaging application, with more than 400 million monthly active users (as of April 2020). Besides user-to-user and group communication, Telegram *channels* are widely adopted by newspapers, financial institutions, and even government agencies for broadcasting official news, in particular during emergency situations. Moreover, Telegram offers an open API to third party developers, allowing for a range of (possibly commercial) services by means of *bots*.

At the core of this ecosystem lies *MTProto 2.0* [19], a suite of cryptographic protocols designed for implementing fast, scalable, *secure* message exchange without relying on the security of the underlying transport protocol. To this end, MTProto is composed by several (sub)protocols handling the initial authentication of the client with the creation of a shared keys between client and server, the creation of shared keys between two clients for end-to-end encryption in secret chats, the rekeying of secret chats, and of course the encryption of all messages, before being transmitted over a (possibly insecure) network.

Although MTProto 2.0 is completely open and client's code is open-source, Telegram's security model has received wide criticism. First and foremost, the choice of the non-standard, *ad hoc* protocol and encryption scheme has been objected, because the lack of scrutiny could expose the system to vulnerabilities, potentially undermining its security [13]. Moreover, all messages (even those of "secret chats") pass through (a cloud of) proprietary, closed-source servers, where they can be stored for any amount of time. This architecture is convenient for users, who can access and synchronise their messages from several devices and send and receive messages also when the peer is not present, but from a security perspective it means that the server must be considered as an *untrusted* party.

This situation raises the need for a practical verification of the MTProto 2.0 protocol suite. However, in spite of the criticisms above, most research has focused on the previous version MTProto 1.0, deprecated since December 2017. To our knowledge, no formal, in-depth verification of MTProto 2.0 has been carried out, so far. This is the scope of this work.

In this paper we formalise and analyse MTProto 2.0 using ProVerif [6], a symbolic cryptographic protocol verifier based on the Dolev-Yao model. In particular, we provide a fully automated proof of the soundness of MTProto 2.0's protocols for authentication, normal chat, end-to-end encrypted chat, and re-keying mechanisms with respect to several security properties, including authentication, integrity, confidentiality and perfect forward secrecy. These properties are verified also in presence of malicious servers.

In order to prove these results we proceed in a modular way. Each protocol of the suite is examined in isolation, specifying which guarantees it requires from previous protocols and which ones it provides; separation between protocols is guaranteed by the typing discipline enforced on messages: "out of sequence" messages are simply discarded. For each protocol we provide its formalisation in ProVerif's specification language (the applied π-calculus), the formalisation of its security properties, and the results of the formal verification.

This modular approach allows us to cope with the complexity of the suite: on one hand the concatenation of these analyses yields the formal correctness of the whole suite; on the other, it allows us to isolate the security properties required on the underlying message encryption scheme. Namely, the only assumption we make is that the latter is an *authenticated encryption scheme*, guaranteeing both integrity of ciphertext (INT-CTXT) and indistinguishability of chosen plaintext (IND-CPA). These properties are difficult to prove in a *symbolic* model like ProVerif's, but can be proved in a *computational* model, e.g. using tools like CryptoVerif or EasyCrypt [5, 2]. This assumption may appear strong, especially considering that Telegram has been widely criticized for its design choices (such as *ad hoc* cryptographic primitives and an unusual encryption mode), and vulnerabilities have been found in MTProto v1.0 (but actually, none of these attacks have been replicated on the new MTProto 2.0). Still, proving the logical correctness of the protocol under a fairly general threat model is very important because, if a weakness in the protocol exists, it must be looked for in the "lower-level" part of the protocol, among the chosen cryptographic functions and other implementation choices. We remark that, thanks to the modular approach, in order to fix any vulnerability that could be found in the message encryption scheme, it will be enough to replace the scheme only, without modifying the protocols. Thus, in this paper we focus on the symbolic verification of MTProto 2.0, leaving the analysis of the encryption scheme in the computational model to future work.

Besides the relevance for Telegram users, the formalisation we present can serve as a reference documentation for the implementations of other MTProto 2.0 clients and servers.

Synopsis. In Section 2 we recall previous related work. The security model adopted in the present work is described in Section 3. In Section 4 we recall the structure of MTProto 2.0. In the successive sections we analyze the subprotocols of MTProto 2.0: initial authorization key creation (Section 5), key exchange for secret chats (Section 6), re-keying in secret chats (Section 7). Conclusions and directions for future work are in Section 8.

We assume the reader confident with ProVerif; for an introduction we refer to [6] and several tutorials online. The code of our formalisation is available at `https://github.com/miculan/telegram-mtproto2-verification`.

2 Related Work

All the published research on MTProto that we are aware of, as well as most online articles, refer to the now deprecated MTProto v1.0 and do not directly apply to the current MTProto 2.0, deployed in Telegram clients as of v4.6 (December 2017).

Arguably, the closest work to ours is [13], where the Signal protocol is formalised in ProVerif. In *loc.cit.* MTProto v1.0 is also briefly discussed, but not at the formal level. The Signal protocol has been studied rigorously in [10, 8, 17], but without a formal verification.

Several issues have been pointed out in MTProto v1.0. Its encryption scheme added a random padding to the message prior to encryption but after the *msg_key* was computed, leading to a couple of theoretical CCA attacks [12, 11]. Besides, earlier versions of the protocol did not provide forward secrecy, and message sequence numbers were managed by the server, so that a malicious server could easily perform replay attacks [12]. Another form of replay attack was discovered in Android's Telegram client v3.13.1 [18], where the same message could be accepted twice by a client after 300 more messages had been sent. This was due to a flaw in the implementation, which did not abide by Telegram's Security Guidelines for Client Developers; in particular, the app did not check that the message ID of the received message was greater than any of the stored IDs. This was fixed in Telegram v3.16 [18].

The above mentioned issues were addressed in MTProto 2.0, which is claimed by its developers to be IND-CCA and INT-CTXT secure and to provide perfect forward secrecy for secret chats. In fact, at the moment no attacks of this kind on MTProto 2.0 are known.

A theoretical MITM attack to MTProto 1.0 has been described in [16]. As we will see in Section 6, the DH exchange used to establish a shared key before initiating a secret chat is not authenticated by the two ends. Clients are supposed to verify a hash of the shared secret through an external secure channel. In MTProto v1.0, the first 128 bits of the SHA1 of the key are used as the fingerprint. A malicious server might social engineer two clients to both initiate a conversation with each other; since the server forwards all the messages, it might act as a MITM and try to find two keys whose fingerprints coincide, using a birthday attack and approximately 2^{65} computations [16]. In MTProto 2.0 (starting with the so-called "layer 46" of secret chat protocol), the fingerprint is 288 bits long (additional 160 bits are extracted from the prefix of the SHA256 of the key), thus making this MITM attack likely infeasible.

3 Security Model

We model Telegram protocols in ProVerif [6], which is a *symbolic* cryptographic verifier. Protocols and security properties are specified in a variant of the *applied* π*-calculus*, a formalism designed for representing cryptographic processes, and translated into a Horn theory. Cryptographic primitives are represented by means of a suitable term theory, by means of constructors and reduction rules or equations; thus, cryptographic primitives are modeled as "perfect", e.g., there is no way to recover a clear text nor the key from ciphertexts.

Following this approach, in our model we consider the message encryption scheme used in MTProto 2.0 as a robust *authenticated-encryption scheme*, abstracting from its actual implementation. An authenticated-encryption scheme, is composed by

- an encryption scheme aenc that takes key k from some fixed keyspace K, a nonce n, a message m and returns a string $C = \mathsf{aenc}_k(n, m)$,
- a decryption scheme adec that takes key k from K and a string C, such that, for all

$$C, k : \mathsf{adec}_k(C) = \begin{cases} m & \text{if } C = \mathsf{aenc}_k(n,m) \text{ for some } n, m \\ error & \text{otherwise.} \end{cases}$$

In practice, adec extracts n from C first (e.g. from the initial part of C), then uses it to decrypt the remaining part of the message.

Formally, in ProVerif the authenticated encryption and decryption primitives are governed by the following reduction rule:

$$\begin{array}{l}\mathsf{fun}\ \mathsf{aenc}(\mathtt{Bitstring}, \mathtt{SharedKey}, \mathtt{Nonce}) : \mathtt{Bitstring}\\ \quad \mathsf{reduc}\ \mathsf{forall}\ m\colon \mathtt{Bitstring}, k\colon \mathtt{SharedKey}, n\colon \mathtt{Nonce}; \mathsf{adec}(\mathsf{aenc}(m, k, n), k) = m.\end{array}$$

A detailed verification of these cryptographic functions in the computational model is left to future work.

Threat Model. We adopt the classical symbolic Dolev-Yao model [9], which is the one used by ProVerif. More specifically, we assume that all messages are transmitted over a public network, and that an active intruder can intercept, modify, forward, drop, replay or reflect any message. Besides, we assume that an attacker may also exfiltrate secret data, such as pre-shared keys, during or after the execution of a protocol. As mentioned above, we assume that encrypted messages are unbreakable unless the key becomes available to the attacker. The model for hash functions is also quite strong, being close to the random oracle model. Timing attacks and guessing attacks are not modeled.

All communication among Telegram clients pass through Telegram servers. Hence, such servers have access to the plaintext of cloud-based chats and to the ciphertext of secret chats. Servers are also responsible for choosing the Diffie-Hellman parameters used to derive clients' long-term authorization keys. Therefore, a server should not be considered as trusted.

Security Goals. Each part of MTProto has different security goals, which we will define in later sections. In general, we will consider the following, informally described, goals:

Secrecy: if a message m is exchanged in a session S between two honest principals A and B then m is kept confidential (i.e., known only to A and B) unless an attacker can break some cryptographic construction or recover the encryption keys before or during S.

Forward secrecy: confidentiality of m is preserved even if the attacker recovers the encryption keys after S is completed.

Authentication: if B receives a message m which is supposed to come from A, then it was really sent by A.

Integrity: if m is sent from A to B then B receives m and not some forged $m' \neq m$ instead.

4 The MTProto 2.0 protocol

In this section we provide a high-level overview of MTProto 2.0. For a deeper (albeit informal) description, we refer the reader to the official web page [19].

MTProto 2.0 is a client/server protocol suite designed for accessing a (MTProto) server[1] from applications running on desktop computers or mobile devices, through an insecure network. This suite can be divided into three main components (see Figure 1):

High-level API and type language: defines how API queries and responses are converted to binary messages. This component fits OSI layers 7 (application) and 6 (presentation).

Cryptographic and authorization components: defines how applications are authenticated with the server, and messages are encrypted before being transmitted through the transport protocol. These components fit OSI layers 5 (session) and 4 (transport).

[1] Actually, Telegram employs a network (a "cloud") of servers in multiple data centers, spread worldwide for scalability and availability. However, for our aims, we can consider this network as a single server.

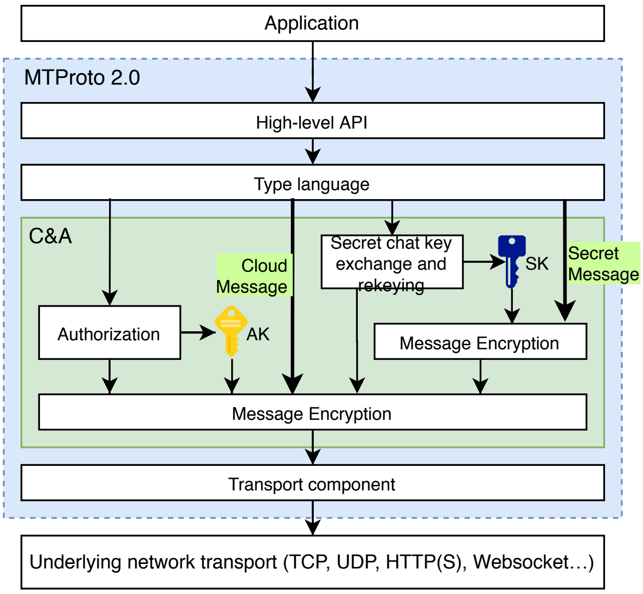

Figure 1 The MTProto 2.0 suite (light blue box). The subject of the present work is the "Cryptography and Authorization" (C&A) component, here represented by the light green box. AK (yellow key) is the Authorization Key, established once at the first run. SK (blue key) is the Secret Chat Session Key, established at the beginning of each secret chat (and changed often). Cloud messages are encrypted only from client to server (and vice versa), with the AK. Secret messages are encrypted twice: with the SK, and then with the AK. In this picture the Message Encryption module is duplicated, but actually it is the same, with different keys.

Transport component: defines how the client and the server actually exchange messages, via some existing transport protocol such as UDP, TCP, HTTP, HTTPS, Websocket over HTTP(S). Notice that also insecure, connectionless protocols are supported.

In this work we focus on the component handling cryptographic transformations and authorization. This component can be divided further in the following modules:

Authorization: this module provides the functionalities for the initial client authorization and server authentication. It is called on the first run of the application, for deriving the *authorization key* (AK), a long-term "master" secret shared with the server only. In order to establish the Authorization Key, this module executes a cryptographic protocol (basically a DH exchange) with the server. We will analyse this protocol in Section 5.

Secret chat key exchange and re-keying: this module provides the functionalities for establishing a session shared secret key (SK) between two clients. It is executed once at the beginning of a secret chat and after 100 exchanged messages between the two parties (or over a week) for installing a new key. In both cases, this module executes a a Diffie-Hellman exchange with the peer client (through the server). We will analyse this protocol in Sections 6 and 7.

Message encryption: All messages between client and server are encrypted with a symmetric cypher, using an ephemeral key derived from the AK. Moreover, messages in secret chats are encrypted also with an ephemeral key derived from the SK. The encryption scheme is the same, but it is use twice (with different keys) in the case of messages in secret chats.

It is important to notice that peer clients never communicate directly: messages always go through the server, where are stored for being retrieved in a second moment. Messages are stored in clear in the case of cloud chats, or encrypted in the case of secure chats.

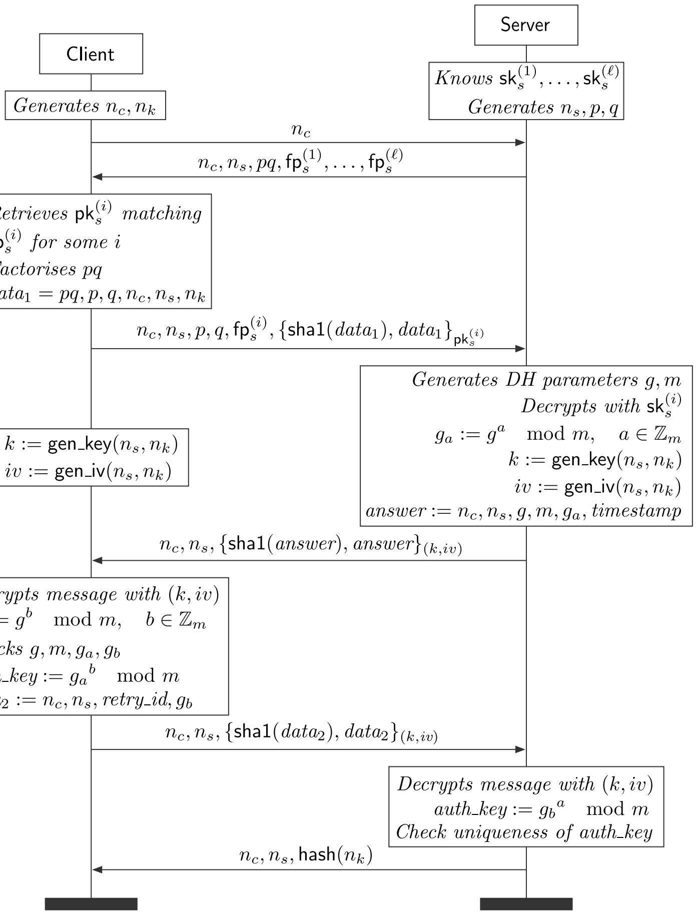
Client
Server
Knows $\mathsf{sk}_s^{(1)}, \dots, \mathsf{sk}_s^{(\ell)}$
Generates n_c, n_k
Generates n_s, p, q
n_c
$n_c, n_s, pq, \mathsf{fp}_s^{(1)}, \dots, \mathsf{fp}_s^{(\ell)}$
Retrieves $\mathsf{pk}_s^{(i)}$ *matching*
$\mathsf{fp}_s^{(i)}$ *for some* i
Factorises pq
$data_1 = pq, p, q, n_c, n_s, n_k$
$n_c, n_s, p, q, \mathsf{fp}_s^{(i)}, \{\mathsf{sha1}(data_1), data_1\}_{\mathsf{pk}_s^{(i)}}$
Generates DH parameters g, m
Decrypts with $\mathsf{sk}_s^{(i)}$
$g_a := g^a \mod m, \quad a \in \mathbb{Z}_m$
$k := \mathsf{gen\_key}(n_s, n_k)$
$iv := \mathsf{gen\_iv}(n_s, n_k)$
$answer := n_c, n_s, g, m, g_a, timestamp$
$k := \mathsf{gen\_key}(n_s, n_k)$
$iv := \mathsf{gen\_iv}(n_s, n_k)$
$n_c, n_s, \{\mathsf{sha1}(answer), answer\}_{(k,iv)}$
Decrypts message with (k, iv)
$g_b := g^b \mod m, \quad b \in \mathbb{Z}_m$
Checks g, m, g_a, g_b
$auth\_key := {g_a}^b \mod m$
$data_2 := n_c, n_s, retry\_id, g_b$
$n_c, n_s, \{\mathsf{sha1}(data_2), data_2\}_{(k,iv)}$
Decrypts message with (k, iv)
$auth\_key := {g_b}^a \mod m$
Check uniqueness of $auth\_key$
$n_c, n_s, \mathsf{hash}(n_k)$

Figure 2 MTProto 2.0's Authentication Protocol. In this and the next diagrams, $\{m\}_{pk}$ represents the asymmetric encryption of m with public key pk, and $\{m\}_{(k,iv)}$ is the symmetric encryption with shared key k and initialization vector iv.

5 Creation of an Authorization Key

On its first run, each Telegram client must negotiate a long-term secret with a Telegram server. Such *authorization key* is created through a Diffie-Hellman (DH) exchange, it is never transmitted over the network, it is used for all subsequent communication between the client C and the server S and it is almost never changed (basically, only when the client application is uninstalled and installed again).

5.1 Informal description

The protocol, shown in Figure 2, consists of three rounds.

Round 1: C and S exchange a pair of randomly generated nonces n_c, n_s, which are sent along with all subsequent messages, both in plaintext and in the encrypted part of each message. Such pair is used to identify a run of the protocol.

1. C sends a nonce n_c to the server.
2. S replies with a message that contains n_c, a fresh nonce n_s generated by S, a challenge $pq < 2^{63}$ (to prevent DoS attacks against Telegram servers) product of two primes p and q, and a list of fingerprints $\mathsf{fp}_s^{(1)}, \dots, \mathsf{fp}_s^{(\ell)}$, with $\ell > 0$, of public keys accepted by S.

Round 2: The client decomposes pq into its factors p and q, and retrieves the public key $\mathsf{pk}_s^{(i)}$ corresponding to some fingerprint $\mathsf{fp}_s^{(i)}$. It also generates a new 256-bit random secret nonce n_k, which, together with the public n_s, is used by both parties to derive a temporary symmetric key k and initialization vector iv via the public functions gen_key and gen_iv (basically, by concatenating hashes of substrings of n_s and n_k). These data are used for encrypting the subsequent messages.

3. C serializes $(pq, p, q, n_c, n_s, n_k)$ and encrypts with $\mathsf{pk}_s^{(i)}$ both this serialized data and its hash; then it sends $n_c, n_s, p, q, \mathsf{fp}_s^{(i)}$ along with the encrypted payload to S.
4. S chooses DH parameters g and m and computes $g_a = g^a \mod m$ for some random 2048-bit number a. Then, S serializes (n_c, n_s, g, e, g_a, t), where t is the server's current time, and encrypts both such serialized data and its hash using a temporary symmetric key k derived from n_s and n_k. It then sends n_c, n_s along with the encrypted payload to C.

Round 3: After deriving k using the same algorithm as the server, C decrypts the received message and checks that these DH parameters are safe (see below). Then C computes $g_b = g^b \mod m$ for some random 2048-bit number b and derives the shared key $g_{ab} = g_a^b \mod m$.

5. C serializes (n_c, n_s, r, g_b), where r is zero at the first attempt to send this message, and it is equal to a hash of the previous authorization key if the server later asks to renegotiate the key (within the same session). C hashes the serialized data and encrypts both the hash and the data using k. It then sends n_c, n_s along with the encrypted payload to S.
6. S derives the shared key as $g_{ab} = g_b^a \mod m$, then verifies that g_{ab} is unique by comparing a hash of g_{ab} to the hashes of already known keys. If the key hash is unique, S sends an acknowledgment $(n_c, n_s, \mathsf{hash}(n_k))$ to C, otherwise S replies with an error message.

All encrypted messages include a SHA1 hash of the content, which the recipient uses after decryption to verify the integrity of the message against network transmission errors. The client is required to check that both m and $(m-1)/2$ are prime, that $2^{2047} < m < 2^{2048}$ and that g generates a cyclic subgroup of prime order $(m-1)/2$. Both parties must also verify that $1 < g, g_a, g_b < m - 1$. Telegram also recommends that both the client and server check that $2^{2048-64} \leq g_a, g_b \leq m - 2^{2048-64}$. Such checks should prevent the use of small subgroups and malicious primes, but it has already been noted that they could be made optional if Telegram used standardized values [13].

5.2 Formalisation in ProVerif

For this protocol, public-key encryption is modelled in the standard way using a reduction of the form $\mathsf{adec}(\mathsf{aenc}(x, \mathsf{pk}(k)), k) = x$, where $\mathsf{aenc}()$ is the encryption function, $\mathsf{adec}()$ is the decryption function and $\mathsf{pk}(k)$ is the public key corresponding to private key k. Thus, the message encryption scheme is assumed to be a secure authentication encryption scheme.

We assume that both parties behave as mandated by the protocol, except for the following misbehaviour:

- a client may fail to verify that the received DH parameters are good, as explained in Section 5.1;
- the server may reuse the same nonce n_s in different sessions.

Client and server process macros are parametrised over their type (good or broken) and misbehaving processes are executed in parallel with correct processes.

Weak DH parameters are modelled as in [3]. The relevant declaration is as follows:

$$\begin{aligned}&\textsf{fun dhExp}(\texttt{Group}, \texttt{Elem}, \texttt{Exp}) : \texttt{Elem}\\ &\quad\textsf{reduc forall } g\colon \texttt{Group}, e\colon \texttt{Elem}, x\colon \texttt{Exp}; \textsf{dhExp}(\texttt{WeakDH}, e, x) = \texttt{BadElem}\\ &\quad\textsf{otherwise forall } g\colon \texttt{Group}, e\colon \texttt{Elem}, x\colon \texttt{Exp}; \textsf{dhExp}(\texttt{StrongDH}, \texttt{BadElem}, x) = \texttt{BadElem}\\ &\quad\textsf{otherwise forall } g\colon \texttt{Group}, e\colon \texttt{Elem}, x\colon \texttt{Exp}; \textsf{dhExp}(\texttt{StrongDH}, e, x) = \textsf{exp}(e, x).\end{aligned}$$

where $\textsf{exp}()$ is governed by the standard equation $\textsf{exp}(\textsf{exp}(g, x), y) = \textsf{exp}(\textsf{exp}(g, y), x)$. In this way, we model a possibly bad choice of parameters by the server by letting the environment (i.e., the attacker) choose them and inject them into the server via the public channel. In other words, the server initially executes $\textsf{in}(c, (g, m))$, where c is the public channel, g is a group generator, and m is a group. A weak calculation always returns the same element, thus conservatively modeling subgroups of size 1. Each computation involving a weak group (symbolized by the constant `WeakDH`) or a bad element (symbolized by the constant `BadElem`) is reduced to the same bad value (`BadElem`). It is worth stressing that other equalities that hold in groups are not modelled (that is difficult or impossible in ProVerif).

Finally, process macros are interleaved with event markers, which can be used to check whether a certain point in a process is reachable (hence, whether a certain event has happened). That allows us to specify some stringent correspondences (see Section 5.3). Events are also used to signal when a secret leaks: for instance, registering and possibly compromising the server's private key is modelled by a process run in parallel with the clients and the server:

$$\begin{aligned}&\textsf{let } \texttt{RSAKeys}() =\\ &\quad\textsf{new } k\colon \texttt{PrivKey};\ \textsf{insert } \texttt{RSAServerKeyTable}(k, \textsf{pk}(k));\ \textsf{out } (c, \textsf{pk}(k));\\ &\quad\textsf{in}(c, \textsf{attack}(=\texttt{ATTACK}));\ \textsf{event } \texttt{CompromisedRSAKey}(k);\ \textsf{out } (c, k).\end{aligned}$$

After registering a key pair (whose private details are inserted into a table owned by the server) and publishing the public key, the private key may be compromised, in which case the corresponding event is recorded. The choice is left to the environment (i.e., the intruder), who can inject the term `attack(ATTACK)` to exfiltrate the key.

5.3 Security properties verification

Authentication. The protocol for generating an authorization key does not prevent an intruder to act as a man-in-the-middle (MITM) during a registration session between a client C and a server S and impersonate C in subsequent exchanges with S. In other words, the protocol does not guarantee the authentication of the client to the server. This is formalized with the following query:

$$\begin{aligned}&\textsf{query } n_c\colon \texttt{Nonce}, n_s\colon \texttt{Nonce};\\ &\textsf{event}(\texttt{ServerAcceptsClient}(n_c, n_s)) \Rightarrow \textsf{event}(\texttt{ClientRequestsDHParameters}(n_c, n_s)),\end{aligned}$$

for which ProVerif can find a counterexample. The query asserts that, if the server accepts a client in a session identified by (n_c, n_s), then it was that client who started session (n_c, n_s).

Failing authentication should not adversely affect the outcome of a session (except that C must possibly restart the protocol in a new session). The only result the intruder could achieve is a negotiation of an authorization key between the intruder and S, unrelated to C.

Vice versa, it is important that C knows with certainty that she has engaged with S and not with an attacker. Assuming that C possesses untampered server's public keys [2] (i.e., the public keys that C has access to really belong to S), authentication of the server for the client is proved by ProVerif. The relevant query is the following:

$$\begin{aligned}&\mathsf{query}\ \mathsf{sk}_s\colon \mathtt{PrivKey},\ n_c, n_s, n_k\colon \mathtt{Nonce},\ g, g_a\colon \mathtt{Elem},\ G\colon \mathtt{Group};\\&\mathsf{event}(\mathtt{ClientReceivesDHParameters}(n_c, n_s, n_k, g, G, g_a))\\&\ \Rightarrow \big(\mathsf{event}(\mathtt{ServerSendsDHParameters}(n_c, n_s, n_k, g, G, g_a))\\&\qquad || \,\mathsf{event}(\mathtt{CompromisedRSAKey}(\mathsf{sk}_s))\,|\,\mathsf{event}(\mathtt{CompromisedNonce}(n_k))\big).\end{aligned}$$

Unless the S's private key sk_s is compromised before or during the session identified by (n_c, n_s) or the secret nonce n_k is leaked, if C accepts Diffie-Hellman parameters (G, g, g_a) in session (n_c, n_s) after sending n_k then C is sure that it was the server who accepted n_k and sent (G, g, g_a) in session (n_c, n_s). This holds even if the server reuses n_s in different sessions, because only S can decrypt n_k and derive the proper temporary key with which DH parameters are transmitted. Note that, since authentication events are registered before C verifies the values received from S, for authentication it does not matter whether the client checks that DH parameters are strong or weak.

Secrecy and forward secrecy. The authorization protocol provides secrecy for the messages subsequently exchanged between the client and the server (in this context, we consider the server a trusted party), which are encrypted using the shared authorization key. Namely, ProVerif proves the following query:

$$\begin{aligned}&\mathsf{query}\ \mathsf{sk}_s\colon \mathtt{PrivKey}, \mathit{auth\_key}\colon \mathtt{SharedKey}, n_k\colon \mathtt{Nonce};\\&\mathsf{attacker}(m) \Rightarrow \big(\mathsf{event}(\mathtt{CompromisedRSAKey}(\mathsf{sk}_s))\,||\,\mathsf{event}(\mathtt{CompromisedNonce}(n_k))\\&||\,\mathsf{event}(\mathtt{ChecksDHParameters}(\bot))\,||\,\mathsf{event}(\mathtt{PostSessionCompromisedKey}(\mathit{auth\_key}))\big).\end{aligned}$$

In words, under the assumptions of our model, the confidentiality of message m is guaranteed unless:

1. the server's private key is compromised before or during the session that establishes the shared authorization key, or
2. the secret nonce n_k generated by the client is leaked during such session, or
3. the client fails to validate the DH parameters received from the server, or
4. the authorization key is compromised at any later time.

This result holds even if the server reuses the same nonce n_s in multiple sessions with different clients. Besides, the result is strict, in the sense that removing any event from the query above leads to a counterexample. Since in our formalization the private key of the server and the secret nonce n_k are always leaked in a separate phase following the completion of the authorization protocol (so that the impact of such leakage of information can be formally assessed), we may also conclude that leaking the server's key or the secret nonce after a session has been completed does not violate the secrecy of subsequent client-server communication. Note also that (4) means that there is no guarantee of forward secrecy for messages encrypted with an authorization key: any previously intercepted message contains a corresponding *msg_key*, which, together with the leaked authorization key, allows an attacker to recover the encrypted payload.

[2] In actual implementations, the public keys of the server are embedded in the application: it is possible that a malicious client embeds a different key without the user to notice.

After two clients have negotiated their authorization keys with a server, they may start to exchange messages within so-called *cloud-based chats*. Every such message is encrypted by the sender using the sender's authorization key and forwarded to the server, who deciphers it and re-encrypts it with the recipient's authorization key. In this context, the server can trivially read (and even modify) every message. The previous result shows that, under the hypothesis that the server is trusted, communication can at least be kept confidential against an external attacker. Cloud-based chats do not provide forward secrecy, though: if, at any time, the authorization key of one of the clients is leaked then all the messages exchanged by that client can be deciphered.

Integrity. The last property we have tried to prove is a basic property of Diffie-Hellman, i.e., key agreement, which can be expressed as follows:

$$\begin{aligned}&\textsf{query}\ n_c, n_s : \texttt{Nonce},\ k, k' : \texttt{SharedKey};\\ &\textsf{event}(\texttt{C-AcceptsAuthKey}(n_c, n_s, k) \,\&\&\, \textsf{event}(\texttt{S-AcceptsAuthKey}(n_c, n_s, k')) \Rightarrow k = k'.\end{aligned}$$

If client and server generate authorization keys k and k' during the same session identified by (n_c, n_s), then they obtain the same key. Unfortunately, ProVerif could neither prove nor disprove such query, even assuming that there are no leaks of private data. A manual inspection of the trace output by ProVerif, however, did not reveal any potential attack. In fact, we do believe that the query holds true.

6 Secret Chats

An end-to-end encrypted chat between two clients A and B can be established after negotiating a session key through a Diffie-Hellman exchange using server S as a forwarder. Each message exchanged between A and B is encrypted with such session key by the sender, then the resulting ciphertext c is in turn encrypted with the sender's long-term authorization key (see Section 5) and sent to S. Both layers of encryption use the same encryption scheme, which we treat symbolically as a cryptographic primitive. Upon receiving a message, S uses the sender's authorization key, uniquely determined based on the 64-bit key fingerprint included in the message to decipher the encrypted payload and recover the ciphertext c, which is then encrypted again with the receiver's authorization key and forwarded to the receiver.

6.1 Informal Description

The protocol, in Fig. 3, is as follows:

1. the initiator A obtains DH parameters (g, p) from S, generates a random session identifier id and a half-key g_a, and sends a request to start an encrypted chat with B including id and g_a in the message;
2. after B has accepted the request, B receives the DH parameters from S and computes a half-key g_b, the shared key g_{ab} and a 64-bit fingerprint $\mathsf{fp}(g_{ab})$ of the key. The values g_b and $\mathsf{fp}(g_{ab})$ are then sent to A, who can compute g_{ab} as well. The fingerprint $\mathsf{fp}(g_{ab})$ is not cryptographically strong: its purpose is only to prevent certain bugs in software implementations, especially during development.

This exchange is unauthenticated, so it is trivial for the server to act as a MITM and establish two different keys with A and B, respectively. To detect such attacks, after the DH exchange

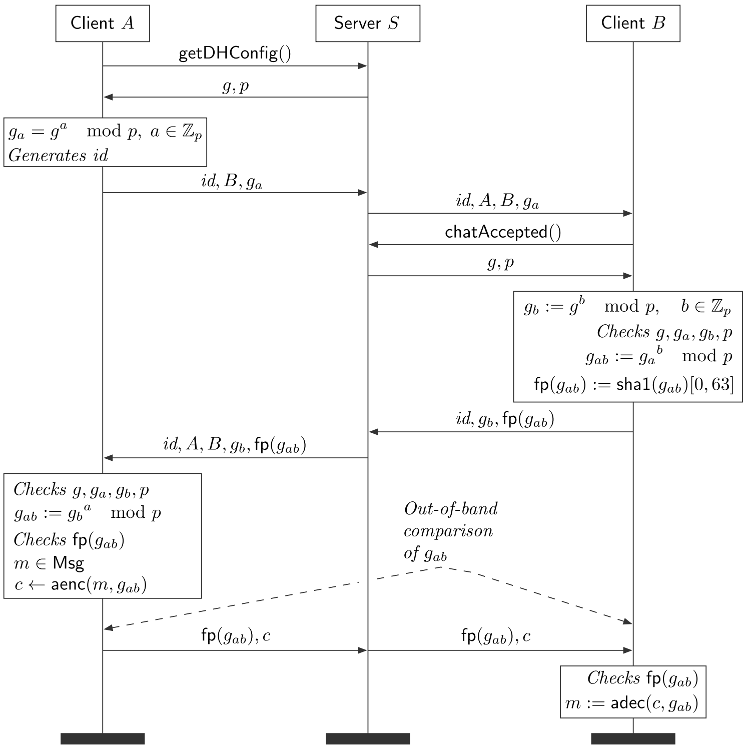
Client A
Server S
Client B
getDHConfig()
g, p
$g_a = g^a \mod p,\ a \in \mathbb{Z}_p$
Generates id
id, B, g_a
id, A, B, g_a
chatAccepted()
g, p
$g_b := g^b \mod p, \quad b \in \mathbb{Z}_p$
Checks g, g_a, g_b, p
$g_{ab} := {g_a}^b \mod p$
$\mathsf{fp}(g_{ab}) := \mathsf{sha1}(g_{ab})[0, 63]$
$id, g_b, \mathsf{fp}(g_{ab})$
$id, A, B, g_b, \mathsf{fp}(g_{ab})$
Checks g, g_a, g_b, p
$g_{ab} := {g_b}^a \mod p$
Checks $\mathsf{fp}(g_{ab})$
$m \in \mathsf{Msg}$
$c \leftarrow \mathsf{aenc}(m, g_{ab})$
Out-of-band comparison of g_{ab}
$\mathsf{fp}(g_{ab}), c$
$\mathsf{fp}(g_{ab}), c$
Checks $\mathsf{fp}(g_{ab})$
$m := \mathsf{adec}(c, g_{ab})$

Figure 3 A slightly simplified version of MTProto 2.0's protocol for secret chats. All messages are forwarded by S: each message between $X \in \{A, B\}$ and S is encrypted using X's authorization key (not shown in the figure). Note that g_{ab}, k and iv are not known to S.

is completed, the clients are required to compare their respective key fingerprints[3] using a secure out-of-band channel. Under this assumption, the protocol should guarantee the secrecy of the messages subsequently exchanged by A and B, which are encrypted using g_{ab} as the shared key. The clients are also supposed to perform suitable checks on the DH parameters, as described in Section 5.

6.2 Formalisation in ProVerif

Each client involved in the secret chat protocol is modelled as a distinct process (an initiator A and a responder B), and communication happens via a common public channel. As the server controls the DH configuration used by the clients to derive their shared key, if the clients do not validate the obtained values then the server might be able to force both clients to use an easily guessable key. For instance, if the server sends a subgroup generator equal to 1 and the clients do not perform any check, the derived session key will be 1 and the server will easily decrypt all the messages.

To model the out-of-band verification that users are asked to perform on a newly generated session key, we use a separate secure channel $\tilde{c}$ available only to A and B, through which a private message $\mathsf{QR}(A, k)$ is sent by A and accepted by B through pattern matching with $\mathsf{in}(\tilde{c}, \mathsf{QR}(=A, =k))$. We assume that clients behave as mandated by the protocol, except for the following deviations:

- a client may fail to verify the DH parameters, as in the authorization protocol (Section 5.2);
- a client may skip the out-of-band validation of the shared key.

In this protocol, the server acts simply as a forwarder and it must be treated as an adversary. Rather than modeling it explicitly as a distinct process, we include the server in the attacker's model by equipping the attacker with the same knowledge as the server (essentially, all the authorization keys) and let the attacker implicitly perform the forwarding. That allows the attacker to receive, manipulate, and resend the exchanged messages in the same way as the server could do, or impersonate a client when the clients do not perform the necessary checks on the received parameters or on the generated key.

6.3 Security properties verification

Secrecy. The main requirement of an end-to-end encrypted chat is, obviously, secrecy: messages exchanged by A and B must be known only to A and B. MTProto's secret chats guarantee secrecy conditional to the strong assumption that clients do validate their keys through a separate private channel. Formally, the secrecy query, which ProVerif is able to prove, can be formulated as follows:

$$\begin{array}{l}\mathsf{query}\ a\colon \texttt{Principal};\\ \mathsf{attacker}(m) \Rightarrow \mathsf{event}(\texttt{ChecksDHConfig}(a, \bot)) \mid\mid \mathsf{event}(\texttt{SkipsKeyCheck}(a, \top)).\end{array}$$

In words, under the assumptions of our model, message m exchanged between two clients is kept confidential unless one of the clients does not perform the mandatory checks of DH parameters or ignores the "manual" authentication of the key via an external secure channel. Note that confidentiality does not rely on the privacy of the authorization keys, i.e., the above query is true even if the authorization keys of the parties involved are leaked before

[3] These are different from $\mathsf{fp}(g_{ab})$; in MTProto 2.0, they are 288-bit hashes that are typically displayed both as hexadecimal strings and QR-codes, suitable for visual comparison, by Telegram clients.

the secret chat protocol starts. Secrecy, however, relies in an essential way on a step that requires active human interaction, at least in the currently available implementations.

Integrity and authentication. The integrity of a message m exchanged by two clients during a secret chat session is also preserved if the clients abide by the rules. The relevant query, proved by Proverif, is as follows:

$$\begin{array}{l}\textsf{query } id, id' : \texttt{ChatID},\ A, B : \texttt{Principal}, k : \texttt{SharedKey},\ m : \texttt{Message};\\ \textsf{inj-event}(\texttt{ReceivesMessage}(id, A, B, m, k)) \Rightarrow \textsf{inj-event}(\texttt{SendsMessage}(id', A, B, m, k))\\ \qquad || \ \textsf{event}(\texttt{ChecksDHConfig}(A, \bot)) \ || \ \textsf{event}(\texttt{ChecksDHConfig}(B, \bot))\\ \qquad || \ \textsf{event}(\texttt{SkipsKeyCheck}(A, k, \top)) \ || \ \textsf{event}(\texttt{SkipsKeyCheck}(B, k, \top)).\end{array}$$

This means that, whenever client B receives a message m that appears to come from A and is encrypted with the shared key k, a message m encrypted with k was indeed previously sent by A and addressed to B, unless one of the clients behaves incorrectly by skipping some essential checks. Note that we cannot prove that the message was sent in the same session in which it is received, because the server might send B a chat id different from the one received from A. This does not seem to pose security risks, though: it is rather a correctness issue related to session management. Anyway, a similar result additionally requiring $id = id'$ can be proved if the clients also compare their respective id's during the out-of-band confirmation step, i.e., if $\mathsf{QR}(id, A, k)$ is sent instead of just $\mathsf{QR}(A, k)$.

7 Re-keying and Perfect Forward Secrecy

The key used in secret chats is replaced every 100 messages or every week (provided that at least one message has been sent) using the protocol shown in Fig. 4. Old keys should be destroyed and never reused.

The exchange uses the same DH parameters obtained when the secret chat was first established (Section 6). The messages are transmitted through the secret channel already in place between the clients, so the server, who acts as a forwarder, can observe only the ciphertexts. As for other client-server communication, each message exchanged between a client and the server is also encrypted using the corresponding authorization key.

7.1 Informal Description

This protocol is simpler to analyse than the previous ones, because it is essentially a run of standard Diffie-Hellman through a secure channel. The claim that the channel is secure relies on the assumptions and results of Section 6.3. Both clients possess the DH parameters (g, p) from their initial run of the secret chat protocol. The new shared key is derived as follows (see Fig. 4):

1. Client A generates a random session id and a random $a \in \mathbb{Z}_p$, computes a half-key g_a and sends the pair (id, g_a) to B.
2. Client B generates a random $b \in \mathbb{Z}_p$ and computes its half-key g_b and the new shared key g_{ab}. The half-key and a fingerprint of the shared key (64 bits of the SHA1 of the key) are sent to A.
3. Client A computes g_{ab}, checks that the fingerprint of g_{ab} matches the received fingerprint, and sends the fingerprint back to B as an acknowledgment.

The previous key is no longer needed and can be deleted.

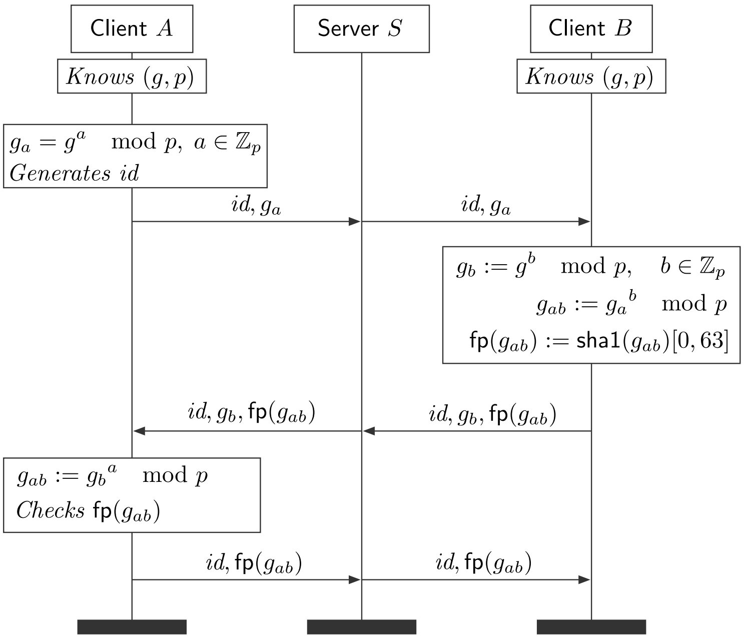
Client A
Server S
Client B
$Knows\ (g,p)$
$Knows\ (g,p)$
$g_a = g^a \mod p,\ a \in \mathbb{Z}_p$
$Generates\ id$
id, g_a
id, g_a
$g_b := g^b \mod p,\quad b \in \mathbb{Z}_p$
$g_{ab} := {g_a}^b \mod p$
$\mathsf{fp}(g_{ab}) := \mathsf{sha1}(g_{ab})[0,63]$
$id, g_b, \mathsf{fp}(g_{ab})$
$id, g_b, \mathsf{fp}(g_{ab})$
$g_{ab} := {g_b}^a \mod p$
$Checks\ \mathsf{fp}(g_{ab})$
$id, \mathsf{fp}(g_{ab})$
$id, \mathsf{fp}(g_{ab})$

Figure 4 The re-keying protocol. Exchanged messages are end-to-end encrypted using the current secret key shared between A and B.

7.2 Formalisation in ProVerif

The formalisation of the re-keying protocol is a textbook implementation of Diffie-Hellman with minor variations. In particular, the computed fingerprints do not play any security role, but are meant only as a sanity check for the implementations. The only important difference is that all the messages exchanged during re-keying are end-to-end encrypted with the current session key. As in secret chats, the server is untrusted and modelled as the attacker.

7.3 Security properties verification

Secrecy and forward secrecy. With the guarantees provided by the analysis of the secret chat protocol (Section 6), messages exchanged between A and B after re-rekeying is completed are kept secret, even if the authorization keys of both parties are compromised before the re-keying protocol is executed. This can be proved in ProVerif by running the re-keying protocol without encrypting the messages with the authorization keys so that they are accessible to the adversary (which models the untrusted server), then letting A and B exchange a message m encrypted with the new key, and finally verifying that the attacker cannot obtain m by testing the following query:

$$\mathsf{query}\ \ \mathsf{attacker}(m).$$

A form of forward secrecy is provided by the periodic rotation of the keys. If a session key is recovered by an attacker, it can be used to decrypt at most 100 messages or a week worth of messages. While older messages cannot be deciphered, in some circumstances this might still be considered an excessive amount of information to leak. Given the above, leaking an authorization key at any time does not compromise the secrecy of any message.

Integrity and authentication. Each party can be confident that the messages received from the other party are authentic if the secret chat protocol (Section 6) has been executed correctly; in particular, if the clients have validated their first session key through a secure channel. Under the perfect cryptography assumption of our model it is also guaranteed that the messages cannot be tampered.

8 Conclusions

In this paper we have presented the formalisation of the MTProto 2.0 protocol suite in the applied π-calculus, and its analysis using the cryptographic protocol verifier ProVerif. This approach adopts the symbolic Dolev-Yao threat mode: an active intruder can intercept, modify, forward, drop, replay or reflect any message. Within this model, we have provided a fully automated proof of the soundness of MTProto 2.0's protocols for first authentication, normal chat, end-to-end encrypted chat, and re-keying mechanisms with respect to several security properties, including authentication, integrity, confidentiality and perfect forward secrecy. These properties are verified also in presence of malicious servers. Our formalization covers also the behaviour of the users, when relevant; for instance, we have proved that if the users do not check the fingerprints of their shared keys, a MITM attack is possible.

In the light of these results, we can affirm that MTProto 2.0 does not present any logical flaw. Vulnerabilities can arise only from the cryptographic primitives, from implementation flaws (e.g. insufficient checks), from side-channels exfiltration (such as timing or traffic analysis), or from incorrect user behaviour. Hence, these are the aspects which deserve further investigation and particular care in the implementation and use of this protocol.

The basic encryption primitive of MTProto 2.0 is assumed to be a perfect authenticated encryption scheme (IND-CCA and INT-CTXT). Although no attack on this scheme is known to date, these properties need to be formally proved in order to deem MTProto 2.0 definitely secure. This proof cannot be done in a symbolic model like ProVerif's, but it can be achieved in a *computational* model, using tools like CryptoVerif or EasyCrypt [5, 2], which we leave to future work. However, even in the very unlikely case that a flaw is found in the encryption scheme, the results in this paper would be still valid: the protocol could be used just by replacing the encryption scheme, and no other changes would be required.

Concerning implementation flaws, our formalisation can be used as a reference for the correct implementation of MTProto 2.0 clients (and servers). Tools like Spi2Java or FS2PV can be useful to this end [4, 15]. Also, particular attention must be paid to side-channel attacks, such as on timing or traffic analysis. A potential issue concerning the correct implementation of clients is about the fact that a server can craft malicious DH parameters, e.g., choosing generators that make discrete logarithms significantly easier to compute [13] or choosing non-primes that pass the 15-round Miller-Rabin test. To prevent the first attack, MTProto prescribes that clients verify that the values received from the server are valid (see Section 5.3). However, as far as we can see, MTProto 2.0 still suffers from the latter vulnerability. A possible improvement is to require clients to check the proposed primes by means of deterministic primality algorithms, such as AKS and Lenstra-Pomerance [1, 14].

Correct user behaviour is crucial in order to prevent MITM attacks in secret chats. As we have seen, to this end users must check the fingerprint of their authorization keys through an external safe channel (actually, this issue concerns not only MTProto 2.0 but any protocol whose keys are defined by means of an insecure DH exchange—including the Signal protocol.) However, it is plausible that in practice such checks are rarely performed, or are performed through the very same (supposedly secure) chat. Hence, users seriously concerned about privacy must be educated about the correct procedure to follow.

References

1 Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes is in p. *Annals of mathematics*, pages 781–793, 2004.

2 Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. In *Foundations of security analysis and design VII*, pages 146–166. Springer, 2013.

3 Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 483–502. IEEE, 2017.

4 Karthikeyan Bhargavan, Cédric Fournet, and Andrew D Gordon. Modular verification of security protocol code by typing. *ACM Sigplan Notices*, 45(1):445–456, 2010.

5 Bruno Blanchet. Cryptoverif: Computationally sound mechanized prover for cryptographic protocols. In *Dagstuhl seminar "Formal Protocol Verification Applied*, volume 117, 2007.

6 Bruno Blanchet. Modeling and verifying security protocols with the Applied Pi Calculus and ProVerif. *Foundations and Trends® in Privacy and Security*, 1(1–2):1–135, 2016.

7 Carl Campbell. Design and specification of cryptographic capabilities. *IEEE Communications Society Magazine*, 16(6):15–19, 1978.

8 Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the Signal messaging protocol. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 451–466. IEEE, 2017.

9 Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

10 Tilman Frosch, Christian Mainka, Christoph Bader, Florian Bergsma, Jörg Schwenk, and Thorsten Holz. How secure is textsecure? In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 457–472. IEEE, 2016.

11 Jakob Jakobsen and Claudio Orlandi. On the CCA (in)security of MTProto. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 113–116, 2016.

12 Jakob Bjerre Jakobsen. A practical cryptanalysis of the Telegram messaging protocol. Master's thesis, Aarhus University, September 2015.

13 Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 435–450. 2017.

14 Hendrik W Lenstra Jr and Carl B Pomerance. Primality testing with gaussian periods. *Journal of the European Mathematical Society*, 21(4):1229–1269, 2019.

15 Davide Pozza, Riccardo Sisto, and Luca Durante. Spi2java: Automatic cryptographic protocol java code generation from spi calculus. In *18th International Conference on Advanced Information Networking and Applications, 2004*, volume 1, pages 400–405. IEEE, 2004.

16 Alex Rad and Juliano Rizzo. A 2^{64} attack on Telegram, and why a super villain doesn't need it to read your telegram chats. (last accessed on April 9, 2020), 2015.

17 Paul Rösler, Christian Mainka, and Jörg Schwenk. More is less: on the end-to-end security of group chats in Signal, Whatsapp, and Threema. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 415–429. IEEE, 2018.

18 Tomáš Sušánka and Josef Kokeš. Security analysis of the Telegram IM. In *Proceedings of the 1st Reversing and Offensive-Oriented Trends Symposium (ROOTS 2017)*, pages 1–8, 2017.

19 Telegram. MTProto mobile protocol. `https://core.telegram.org/mtproto/` (last accessed on May 3rd, 2020), 2020.

20 Telegram. Telegram FAQ. `https://telegram.org/faq` (last accessed on April 9th, 2020).