Automated Symbolic Verification of Telegram's MTProto 2.0

Marino Miculan*[a] and Nicola Vitacolonna[b]

Department of Mathematics, Computer Science and Physics, University of Udine, Italy

Abstract: *MTProto 2.0* is a suite of cryptographic protocols for instant messaging at the core of the popular Telegram messenger application. In this paper we analyse MTProto 2.0 using the symbolic verifier ProVerif. We provide fully automated proofs of the soundness of MTProto 2.0's authentication, normal chat, end-to-end encrypted chat, and rekeying mechanisms with respect to several security properties, including authentication, integrity, secrecy and perfect forward secrecy; at the same time, we discover that the rekeying protocol is vulnerable to an *unknown key-share* (UKS) attack. We proceed in an incremental 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 w.r.t. most relevant security properties, and it can serve as a reference for implementation and analysis of clients and servers.

1 INTRODUCTION

Telegram is a very popular instant messaging application, with more than 500 million active users (as of March 2021). Besides user-to-user and group communication, *channels* are widely adopted by newspapers, financial institutions, and even government agencies for broadcasting official news, in particular during emergency situations. Telegram provides also an open API allowing third parties to offer (possibly commercial) services by means of *bots*.

At the core of this ecosystem lies *MTProto 2.0* (Telegram, 2021a), 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 clients with the creation of shared keys between clients and server, the creation of session 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 fore-

most, 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 (Kobeissi et al., 2017). 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 (Blanchet, 2016), a state-of-the-art symbolic cryptographic protocol verifier based on the Dolev-Yao model. 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 rekeying mechanisms with respect to several security properties, including authentication, integrity, secrecy and perfect forward secrecy. These properties are verified also in presence of mali-

[a] https://orcid.org/0000-0003-0755-3444
[b] https://orcid.org/0000-0002-7505-5124

cious servers and clients. On the other hand, we discover that, in principle, the rekeying protocol is vulnerable to an *unknown key-share* (UKS) attack: a malicious client E can induce two honest clients A, B to believe they share two secret keys with E, and instead they share the same key between themselves only.

In order to prove these results we proceed in an incremental 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 approach allows us to cope with the complexity of the suite and 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 (Blanchet, 2007; Barthe et al., 2013). This assumption may appear strong, especially considering that Telegram has been widely criticized for its design choices, and vulnerabilities have been found in MTProto 1.0; however, none of these attacks have been replicated on MTProto 2.0.

In this paper we focus on the symbolic verification of MTProto 2.0 with respect to an unbounded number of sessions, leaving the analysis of the encryption scheme in the computational model to future work. Proving the logical correctness of a protocol under a fairly general threat model is very important: if a protocol is proved to be sound at a symbolic level then one can focus on searching vulnerabilities in the "lower" part of the protocol stack, among the chosen cryptographic functions and other implementation choices. If a cryptographic primitive is found to be weak, it can be substituted with a stronger one, without invalidating the symbolic analysis.

Besides the relevance for Telegram's users, our formalisation can serve as a reference documentation for the implementation of clients and servers.

Synopsis. In Section 2 we recall previous related work. The security model adopted is described in Section 3. In Section 4 we recall the structure of MTProto 2.0, whose protocols are analyzed in the subsequent sections: initial authorization key creation (Section 5), key exchange for secret chats (Section 6), rekeying 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 (Blanchet, 2016) and several tutorials online. The code 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 (Kobeissi et al., 2017), where the Signal protocol is formalised in ProVerif and MTProto v1.0 is also briefly discussed, but not at the formal level. The Signal protocol has been studied rigorously in (Frosch et al., 2016; Cohn-Gordon et al., 2017; Rösler et al., 2018).

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 (Jakobsen and Orlandi, 2016). 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 (Jakobsen, 2015). Another replay attack was discovered in Android's Telegram client v3.13.1 (Sušánka and Kokeš, 2017), 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; it was fixed in Telegram v3.16.

The above mentioned issues were addressed in MTProto 2.0, whose developers claim to be IND-CCA and INT-CTXT secure and to provide perfect forward secrecy for secret chats. There is currently no known proof of such claims, but there are no known attacks on MTProto's encryption, either.

A theoretical MitM attack to MTProto 1.0 has been described in (Rad and Rizzo, 2015). 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 (Rad and Rizzo, 2015). In MTProto 2.0 (starting with the "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 (Blanchet, 2016), 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"; in particular, there is no way to recover a plaintext or a key from the ciphertexts, and the ciphertext is not forgeable.

Following this approach, in our model we consider the message encryption scheme used in MTProto 2.0 essentially as an authenticated encryption scheme, abstracting from its actual implementation.

Formally, MTProto's symmetric encryption primitive is governed by the following reduction rule:

$\mathsf{fun}\ \mathsf{senc}(\mathsf{Bitstring}, \mathsf{SharedKey}, \mathsf{Nonce}) : \mathsf{Bitstring}$
$\quad \mathsf{reduc}\ \mathsf{forall}\ m : \mathsf{Bitstring}, k : \mathsf{SharedKey}, n : \mathsf{Nonce};$
$\quad\quad \mathsf{sdec}(\mathsf{senc}(m,k,n),k) = m.$

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 (Dolev and Yao, 1983), 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. Other threats, such as timing attacks and guessing attacks, are beyond the scope of our model.

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

Security Goals. Each part of MTProto has different security goals, as we will see later on. In general, we will consider the following, informally stated, goals:

Secrecy: if a message m is exchanged in a session S between two honest principals A and B then m is secret (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: secrecy of message m is preserved even if the attacker recovers the encryption keys after S is completed.

Authentication: if B receives 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 give a high-level overview of MTProto 2.0. For a deeper (albeit informal) description, we refer to the official web page (Telegram, 2021a).

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 Fig. 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).

Transport Component: defines how the client and the server actually exchange messages, via existing transport protocol such as UDP, TCP, HTTP, HTTPS, Websocket, etc.. Note that also insecure, connectionless protocols are supported.

---

[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.
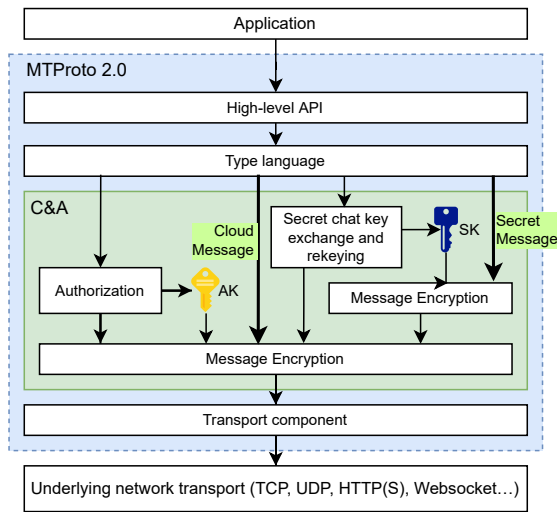
Figure 1: The MTProto 2.0 suite. The subject of the present work is the "Cryptography and Authorization" (C&A) component. AK (yellow key) is the Authorization Key, established once at the first run. SK (blue key) is the Secret Chat Session Key, established at the beginning of each secret chat (and changed 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.

We focus on the component handling cryptographic transformations and authorization. This component can be divided further into 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 Rekeying: 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 cipher, using an ephemeral key derived from the AK. Messages in secret chats are encrypted also with an ephemeral key derived from the SK. The
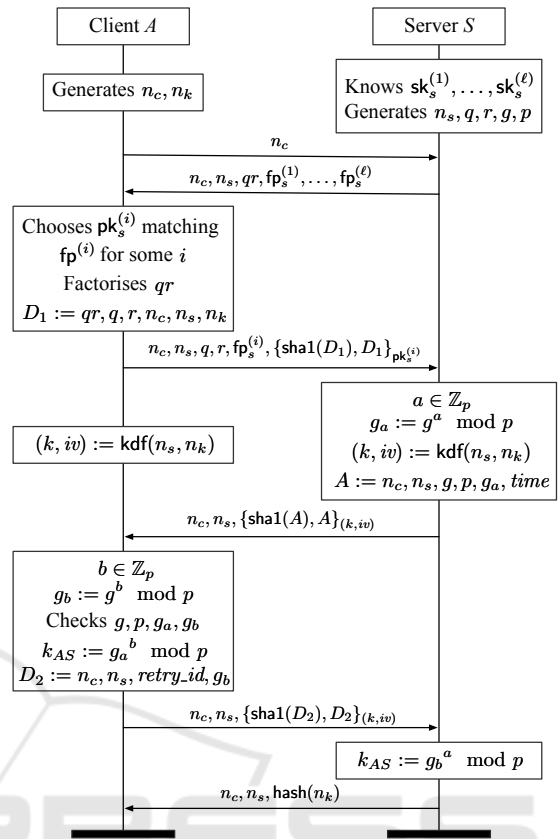

Figure 2: MTProto 2.0's Authorization Protocol.

encryption scheme is the same, but it is use twice (with different keys) for messages in secret chats.

We stress that peer clients never communicate directly: messages always go through a server, where they are stored to permit later retrieval by the recipient. Cloud chat messages are kept in clear text, while secret chat messages are encrypted with the peers' session key, which should be unknown to the server.

5 AUTHORIZATION KEYS

On its first run, each Telegram client A must negotiate a long-term secret with a Telegram server S. Such an *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 A and S, and it is almost never changed (basically, only when the client application is uninstalled and installed again).

5.1 Informal Description

The protocol consists of three rounds (Fig. 2):

Round 1. *A* and *S* exchange a pair of randomly generated nonces n_c and 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. *A* sends a nonce n_c to *S*.

2. *S* replies with a message that contains n_c, a fresh nonce n_s generated by *S*, a challenge $qr < 2^{63}$ (to prevent DoS attacks against Telegram servers), which is the product of two primes q and r, and a list $fp_s^{(1)}, \ldots, fp_s^{(\ell)}$, with $\ell > 0$, of the fingerprints of the public keys accepted by *S* (computed as the 64 lower bits of the SHA-1 of the keys).

Round 2. *A* decomposes qr into its factors q and r, and retrieves[2] the public key $pk_s^{(i)}$ corresponding to some fingerprint $fp_s^{(i)}$. 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 an ephemeral symmetric key k and initialization vector iv by hashing some substrings of n_s and n_k. These data are used for encrypting the subsequent messages.

3. *A* serializes $(qr, q, r, n_c, n_s, n_k)$ and encrypts with $pk_s^{(i)}$ both this serialized data and its hash; then, it sends $n_c, n_s, q, r, fp_s^{(i)}$ along with the encrypted payload to *S*.

4. *S* chooses DH parameters g and p and computes $g_a = g^a \mod p$ for some random 2048-bit number a. Then, *S* serializes (n_c, n_s, g, p, g_a, t), where t is the server's current time, and encrypts both the 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 *A*.

Round 3. After deriving (k, iv) using the same algorithm as the server, *A* decrypts the received message and checks that the received DH parameters are safe (see below). Then *A* computes $g_b = g^b \mod p$ for some random 2048-bit number b and derives the authorization key $k_{AS} = g_a{}^b \mod p$.

5. *A* serializes $(n_c, n_s, retry\_id, g_b)$, where *retry_id* is 0 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 by generating a new b (the server might ask the client to do this for the failure of the uniqueness check performed at the end of the protocol—see next paragraph). *A* hashes the serialized data and encrypts both the hash and the

---

[2]It is assumed that the server's public keys are known to the clients. In practice, they are usually embedded in Telegram's client apps. Of course, it is possible that a malicious client embeds a different key without the user to notice.

data with (k, iv). It then sends n_c, n_s along with the encrypted payload to *S*.

6. *S* derives the shared key $k_{AS} = g_b{}^a \mod p$, then verifies that k_{AS} is unique by comparing a hash of k_{AS} to the hashes of all authorization keys known to the server. If the hash is unique, *S* sends an acknowledgment $(n_c, n_s, hash(n_k))$ to *A*, otherwise *S* sends an error message.

All the encrypted messages include a SHA1 hash of the content. According to Telegram's "Advanced FAQ" (Telegram, 2021b), such hashes are "irrelevant for security", so they are presumably used only an additional sanity check for the implementation, and not as a means of implementing an AEAD (authenticated encryption with additional data) scheme.

The client is required to check that both p and $(p-1)/2$ are prime, that $2^{2047} < p < 2^{2048}$ and that g generates a cyclic subgroup of prime order $(p-1)/2$. Both parties must also verify that $1 < g, g_a, g_b < p-1$. Telegram also recommends that both the client and server check that $2^{2048-64} \leq g_a, g_b \leq p - 2^{2048-64}$. 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 (Kobeissi et al., 2017).

5.2 Formalisation in ProVerif

In the formalization of the protocol we made a few simplifications: we ignored *time* (relevant only to correctly generate message identifiers in later communication, which we do not model because we consider messages abstractly), *retry_id* (thus modeling explicitly only a successful path of execution—failures are implicitly taken account of by running an unbounded number of sessions, some of which may not complete), the proof-of-work decomposition of qr (because DoS attacks are not in our threat model) and, more importantly, the SHA1 hashes inside the encrypted messages. Including such aspects into a symbolic model would significantly increase the computation times required for the automated verification, but it would most likely not allow us to discover more vulnerabilities. The latter simplification, in particular, does not affect ProVerif's results (but it improves verification times), because in the symbolic model encryption is already authenticated, so adding a MAC is not really needed; besides, that aligns with Telegram's statement of "irrelevance" of such tags for security.

For this protocol, public-key encryption is modelled in the standard way using a reduction of the form $adec(aenc(x, pk(k)), k) = x$, where $aenc()$ is the encryption function, $adec()$ is the decryption function and $pk(k)$ is the public key corresponding to pri-

vate key k. Thus, the message encryption scheme is assumed to be a secure authentication encryption scheme. We assume that honest 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; and the server is allowed to reuse the same nonce n_s in different sessions. Misbehaving processes are executed in parallel with correct processes.

Weak DH parameters are modelled as in (Bhargavan et al., 2017). The generation of (possibly weak) DH parameters is delegated to two parallel processes as explained below. A weak calculation always returns the same element, thus conservatively modeling subgroups of size 1. Each computation involving a weak group or a bad element is reduced to the same bad value. It is worth stressing that other equalities that hold in groups are not modelled, although this model is stronger than the Diffie-Hellman assumption, in the sense that is impossible to compute g^{xy} given only generator g, g^x and g^y.

Process macros are interleaved with event markers ("event $P(\vec{x})$" for some predicate $P(\vec{x})$), 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 is compromised. Leaking information is modelled via several short processes that run in parallel with the clients and the server. For instance, compromising the server's private key is achieved by running a process that receives the server's private key from the server through a private channel c_{priv} and makes it accessible to the attacker through the public channel c, recording the event:

$$\text{let} \quad \text{LeakRSAKey}() = $$
$$\text{in}(c_{priv}, k : \text{PrivKey});$$
$$\text{event CompromisedRSAKey}(k);$$
$$\text{out}(c, k).$$

Similar parallel processes are executed to perform the following functions and emit the corresponding event:

- CompromisedNonce(n_k): leaking the secret nonce n_k generated by the client during a session;

- PostCompromisedNonce(n_k): revealing n_k after the protocol is completed;

- PostCompromisedRSAKey(k): revealing the server's private key k after the protocol is completed;

- PostCompromisedAuthKey(k): revealing the authorization key k after the protocol is completed (to test forward secrecy);

- ForgedServerIdentity(S): associating the server's identity S to a private key chosen by the attacker.

Other four parallel processes are responsible for generating a random or fixed server nonce n_s, respectively, and for generating good or bad DH parameters, respectively, on behalf of the server (by communicating with the server through private channels). In general, we found that using a higher number of relatively simple processes rather than a smaller number of parametrized processes with a more complex logic allows ProVerif to perform much better, and gives the user more flexibility in expressing queries.

5.3 Security Properties Verification

Authentication. The protocol for generating an authorization key does not prevent an intruder to act as a MitM during a registration session between a client A and a server S and impersonate A in subsequent exchanges with S. In other words, the protocol does not guarantee the authentication of the client to the server—which is expected, as the server is willing to run the protocol with any entity. This is formalized with the following query:

query n_c : Nonce, n_s : Nonce;
event(ServerAcceptsClient(n_c, n_s))
$\quad \Rightarrow$ event(ClientRequestsDHParameters(n_c, n_s)),

for which ProVerif can find a counterexample even when the protocol is run correctly and without any information leak. 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) —which may not be the case, because an attacker can take over after the first round and replace the legitimate client.

Failing authentication should not adversely affect the outcome of a session (except that A must possibly restart the protocol in a new session). The only result the intruder could achieve is a negotiation of an authorization key with S, unrelated to A. Vice versa, it is important that A knows with certainty that she has engaged with S and not with an attacker. Authentication of the server to the client is proved by ProVerif; the relevant query is the following:

query sk_s : PrivKey, n_c, n_s, n_k : Nonce, g, g_a : G
  inj-event(ClientReceivesDHParameters(n_c, n_s, n_k, g, g_a))
  $\Rightarrow$ inj-event(ServerSendsDHParameters(n_c, n_s, n_k, g, g_a))
    $\lor$ event(CompromisedRSAKey(sk_s))
    $\lor$ event(CompromisedNonce(n_k))
    $\lor$ event(ForgedServerIdentity()).

The query is interpreted as follows: unless 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

during the session, or the server's identity is forged to begin with (e.g., the client embeds a spoofed public key), if A accepts Diffie-Hellman parameters (g, g_a) in session (n_c, n_s) after sending n_k then A is sure that it was the server who accepted n_k and sent (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 ephemeral key with which DH parameters are transmitted. Note that, since authentication events are registered before A verifies the values received from S, for authentication it does not matter whether the client checks that DH parameters are strong or weak. The last disjunct reflects the fact that Telegram, as is common for current messenger apps, is based on a Trust-On-First-Use (TOFU) authentication infrastructure.

Secrecy and Forward Secrecy. The authorization protocol provides secrecy for so called "cloud messages", i.e, the messages subsequently exchanged between the client and the server, which are encrypted using the shared authorization key. Namely, ProVerif proves the following query:

query sk_s : PrivKey, k : SharedKey, n_k : Nonce, m : Msg;
    attacker(m)
        $\Rightarrow$ event(CompromisedRSAKey(sk_s))
          $\lor$ event(ForgedServerIdentity())
          $\lor$ event(CompromisedNonce(n_k))
          $\lor$ event(ChecksDHParameters($\perp$))
          $\lor$ event(PostCompromisedAuthKey(k)).

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

1. the server's private key is compromised or its identity is forged 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 a session (in particular, if the attacker learns n_k before the fourth message then it can compute (k, iv) and act as a MitM to establish two distinct authorization keys with the client and the server, respectively), 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 encrypted with the authorization key. Intuitively, this holds because, even if such post-compromise allows an attacker to decrypt all the messages exchanged during the definition of the authorization key, she cannot compute the authorization key knowing only g, m, g_a, g_b (assuming that the DH problem is hard). Note also that case 4 above means that there is no guarantee of forward secrecy for messages encrypted with an authorization key.

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 secret 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. Key agreement, a basic property of Diffie-Hellman, can be proved by ProVerif, using the following query:

query n_c, n_s, n_k : Nonce, k, k' : SharedKey, sk_s : PrivKey;
    event(ServerAcceptsAuthKey(n_c, n_s, k))
    $\land$ event(ClientAcceptsAuthKey(n_c, n_s, k'))
    $\Rightarrow k = k'$
        $\lor$ event(ClientChecksDHParameters($\perp$))
        $\lor$ event(ForgedServerIdentity())
        $\lor$ event(CompromisedNonce(n_k))
        $\lor$ event(CompromisedRSAKey(sk_s)).

If client and server generate authorization keys k and k' during the same session identified by (n_c, n_s), then they compute the same key, unless the client fails to validate the DH parameters, the server's identity is forged, or some secret is leaked during the session.

Conversely, if client and server end a run of the protocol agreeing on the same key in their respective sessions, then such sessions coincide, unless the client skips its mandatory checks. That is, the following query is proved by ProVerif:

query n_c, n'_c, n_s, n'_s: Nonce, k: SharedKey;
event(ServerAcceptsAuthKey(n_c, n_s, k))
$\wedge$ event(ClientAcceptsAuthKey(n'_c, n'_s, k))
$\Rightarrow (n_c = n'_c \wedge n_s = n'_s)$
$\vee$ event(ClientChecksDHParameters($\bot$)).

The last disjunct accounts for the fact that, if the server sends the same bad DH parameters to a client and to the attacker (acting as a malicious client) in two separate sessions and the honest client does not reject them, then the client, the attacker and the server may end up with the same key.

6 SECRET CHATS

An end-to-end encrypted chat between two clients A and B can be established after negotiating a session key k through a Diffie-Hellman exchange using server S as a forwarder. Each message exchanged between A and B is encrypted with k 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 use a similar encryption scheme, which we treat symbolically as a cryptographic primitive. Upon receiving a message, S uses the sender's authorization key, identified by 64-bit key fingerprint prepended to 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, shown in Fig. 3, is as follows:

1. initiator A obtains DH parameters (g, p) from S, generates a random session identifier id and a half-key g_a, and requests 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 session key k and a 64-bit fingerprint hash(k) of the key. The values g_b and hash(k) are then sent to A, who can compute k as well. The fingerprint hash(k) is not cryptographically strong: its stated 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 is completed, the clients
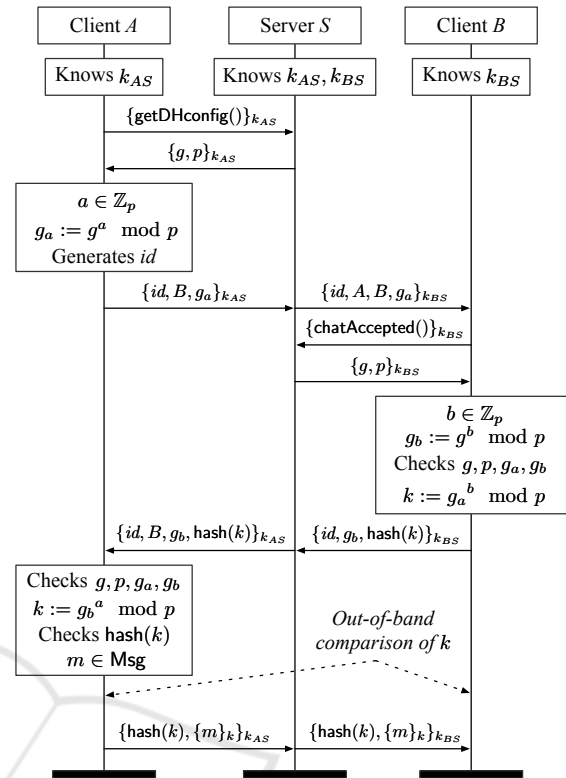

Figure 3: MTProto 2.0's protocol for secret chats.

are required to compare their respective key fingerprints[3] using a secure authenticated out-of-band channel. Under this assumption, the protocol should guarantee the secrecy of the messages subsequently exchanged by A and B, encrypted with k. The clients are also supposed to perform suitable checks on the DH parameters, as described in Section 5. Note that, in actual implementations, the out-of-band check is left to users, i.e., it is not done automatically.

6.2 Formalisation in ProVerif

Each (honest) 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 (without encrypting messages with authorization keys), which allows us to treat the server as an adversary equipped with the same knowledge as the server (essentially, all the authorization keys) and implicitly performing message 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

---

[3]These are different from hash(k); 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.

client if the clients do not perform the required checks on the received parameters or on the generated key.

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 sub-group 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, to B, and to two auxiliary processes, one performing the required validation on behalf of the clients, and one skipping it:

```
let PerformOutOfBandKeyComparison() =
   in(c̃, QR(X,Y,k));
   in(c̃, QR(= Y, = X, = k));
   event OutOfBandKeyComparisonSucceeded(X,Y,k);
   event OutOfBandKeyComparisonSucceeded(Y,X,k);
   out(c̃, QR_OK(X,Y,k));
   out(c̃, QR_OK(Y,X,k)).

let SkipOutOfBandKeyComparison() =
   in(c̃, QR(X,Y,k));
   event OutOfBandKeyComparisonSkipped(X,k);
   out(c̃, QR_OK(X,Y,k)).
```

Each client X wishing to validate a session key negotiated with Y sends a private message $\mathsf{QR}(X,Y,k)$ via $\tilde{c}$ and waits for $\mathsf{QR}_{OK}(X,Y,k)$ as a confirmation. The first of the two processes above waits for a $\mathsf{QR}()$ message from a client and then expects a matching message from the other client. Only when the second message arrives, the process proceeds by recording the event and sending out the confirmation messages. The second process simply acknowledges a $\mathsf{QR}()$ message and records the corresponding event, without doing anything else. Note that, in our model, out-of-band key validation implies mutual authentication. How such channel is to be established in the real world, however, is not specified by the protocol.

Finally, we assume that honest clients behave as per the protocol, except for the following deviations: a client may fail to verify its DH parameters, as in the authorization protocol (Section 5.2), and a client may skip the out-of-band validation of the session 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:

```
query X : Principal, k : SessionKey, m : Msg;
   attacker(m)
     ⇒ event(OutOfBandKeyComparisonSkipped(X,k))
       ∨ event(ClientChecksDHConfig(X,⊥)).
```

In words, under the assumptions of our model, message m exchanged between two clients is kept secret unless one of the clients does not perform the mandatory checks of DH parameters or the clients omit the "manual" authentication of the shared key via an external secure channel.[4] Note that secrecy 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 the secret chat protocol starts. Secrecy, however, crucially relies on a step that requires active human interaction, at least in the current implementations.

As for the confidentiality of the session key k, note that although in a correct run of the protocol k remains secret (otherwise the attacker could obtain m), k is not indistinguishable from a random key: by comparing $\mathsf{choice}[k,r]$ (i.e., one between the real key k and a random value r) with the hash from the last message of the protocol, the server can tell apart k from r with high probability. Accordingly, ProVerif cannot prove the corresponding observational equivalence.

Integrity and Authentication. The integrity and authenticity of a message exchanged by two clients during a secret chat session is preserved if the clients abide by the rules—in particular, if they perform the required out-of-band key validation in a way that implies mutual authentication. But, even under such assumptions, the protocol remains vulnerable to a limited *cross-session attack*. The strongest result we could prove in ProVerif is the following:

```
query i, i′ : ChatID, X,Y,I,R : Principal,
   k : SessionKey, m : Message;
inj-event(ReceivesSecretChatMsg(X,i,I,R,k,m))
  ⇒ inj-event(SendsSecretChatMsg(Y,i′,I,R,k,m))
  ∨ inj-event(SendsSecretChatMsg(Y,i′,R,I,k,m))
  ∨ (event(OutOfBandKeyComparisonSkipped(X,k))
    ∧ event(OutOfBandKeyComparisonSkipped(Y,k)))
```

---

[4]Note that in our formalization (see Section 6.2), it is not possible for one client to validate the session key and for the other to skip the validation.

$$\vee \big(\mathrm{event}(\mathsf{ClientChecksDHConfig}(X, \bot))$$
$$\wedge\, \mathrm{event}(\mathsf{ClientChecksDHConfig}(Y, \bot)) \big).$$

In this query, I denotes the chat initiator and R is the responder. If the clients behave correctly, whenever an honest party X (which, by the definition of the events, may be only I or R) receives a message m encrypted with the session key k in a chat session initiated by I with intended partner R, exactly one message m encrypted with k has indeed been sent by Y (which, again, can only be I or R) in a session started by I with R *or* in a parallel session started by R with I. The latter is due to the possibility that the server S impersonates the responder in two distinct sessions, one started by X with Y, and another started by Y with X, performing the following exchange:

Session 1 $X \to S$: g^x

Session 2 $Y \to S$: g^y

Session 2 $S \to Y$: g^x

Session 1 $S \to X$: g^y

The server impersonating Y in Session 1 and X in Session 2 gets X and Y to compute the same key g^{xy} in both sessions. The server is then free to forward messages across such sessions: they are accepted because they are encrypted with a valid key. Note that this attack is limited to the same two parties X and Y as long as the clients validate their keys.

In the query above we cannot require $i = i'$, because the server might forward X and Y different chat id's. 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 $i = i'$ can be proved if the clients also compare their respective id's during the out-of-band confirmation step, i.e., if $\mathsf{QR}(i, X, Y, k)$ is sent instead of just $\mathsf{QR}(X, Y, k)$.

If the message is not sent by Y, then X and Y did not validate k, or both clients skipped the needed checks on the DH parameters. The former enables the server to perform the classical MitM attack on Diffie-Hellman. The latter allows the server to mount a MitM attack similar to the one described in the last paragraph of Section 5.3. Besides, it is necessary to assume that out-of-band validation really implies mutual authentication: that is the case in our model, but depending on how such validation is carried out in practice, such an assumption may be too strong. For instance, a malicious client E could "social-engineer" two honest clients A and B to make them believe that their key k match E's (who does not have k), e.g., E might get the key fingerprint from B and hand it to A when asked by A to compare their keys.
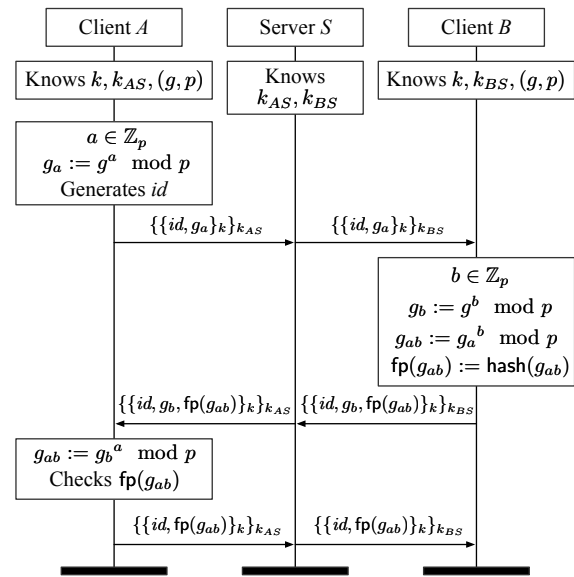

Figure 4: The rekeying protocol.

7 REKEYING

Keys used in secret chats are replaced every 100 messages or every week (if 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). 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.

7.1 Informal Description

The rekeying protocol is essentially a standard Diffie-Hellman exchange over a secure channel. The claim that the channel is secure is only relative to 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 and a shared key k. 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 old key is no longer needed and can be deleted.

7.2 Formalisation in ProVerif

The formalisation of the rekeying protocol is similar to the formalization of the secret chat protocol, using a process for the initiator and one for the responder, while the server is untrusted and included in the attacker. The computed fingerprints are not cryptographically strong and do not play any security role, but are meant only as a sanity check for the implementations. The only important difference compared to a standard Diffie-Hellman exchange is that all the messages exchanged during rekeying are end-to-end encrypted with the current session key k, which is assumed to be known only to the clients.

7.3 Security Properties Verification

Secrecy and Forward Secrecy. Assuming that the current k is strong and secret, secrecy after rekeying is preserved even if the authorization keys of both parties are compromised before the rekeying protocol is executed. This can be proved in ProVerif by running the rekeying 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:

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

Forward secrecy and future secrecy are guaranteed 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 or newer messages cannot be deciphered, in some circumstances such window of compromise might still be considered excessively wide. Given the above, leaking an authorization key at any time does not compromise the secrecy of any message.

Authentication. If A, B run the rekeying protocol negotiating a new key k, then both A and B, and only them, should know k. Unfortunately, the rekeying protocol fails to provide *implicit key confirmation* (the guarantee to A that B can compute k, and vice versa) and *implicit key authentication* (the guarantee to A that no one but B knows k, and vice versa) (Blake-Wilson and Menezes, 1999). A rogue client E, who has already completed the secret chat protocol with A and B, may unfold the following attack:

- A starts rekeying with E and sends its half-key g_a;
- E starts a parallel session with B, sending g_a to B;
- B sends E its half-key g_b;

- E sends A the half-key g_b obtained from B.

At this point, A and B compute the same key, and each one mistakenly believes that the key is shared with E. The following query is disproved by ProVerif, even assuming that the DH parameters are good:

query i : SessionID, I, I', R, R' : Principal, k : SessionKey;
$\quad$ event(InitiatorNegotiatesNewKey(i, I, R, k))
$\quad \wedge$ event(ResponderNegotiatesNewKey(i, I', R', k))
$\quad \Rightarrow I = I' \vee I = R' \vee R = I' \vee R = R'.$

If the initiator I accepts k from R and responder R' accepts k from I', ideally only two parties should be involved (the query is even weaker than that, asking only that at least one entity appears in both events).

As an example of how such a vulnerability could be exploited, consider the following *game of spies*: Alice and Eve are agents from Eurasia, and Bob is an agent from Eastasia. Alice trusts Eve and would never communicate with Bob, but Eve has been paid by Bob to perform the above attack. After the rekeying exchange described above, Bob would be able to transmit misleading information to Alice, who would accept it as coming from Eve. The main advantage compared to having Eve act as a forwarder is that Bob does not need to rely on Eve in subsequent communication; it is also a less risky strategy for Eve (she could eventually claim that her device has been hacked).

Telegram's documentation recommends that clients do not start two rekeying instances concurrently. But the query above generates a trace involving four parties A, B, and two cooperating malicious clients E, E', in which A runs rekeying with E, and E' runs rekeying with B. Each client runs a single instance of the protocol. So, the above recommendation is not a mitigation for this attack.

8 CONCLUSIONS

We have presented the formalisation of the MTProto 2.0 protocol suite in the applied π-calculus, and its analysis using the protocol verifier ProVerif. This approach adopts the symbolic Dolev-Yao threat model: 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 rekeying mechanisms with respect to several security properties, including authentication, integrity, secrecy and perfect forward secrecy, also in the presence of malicious servers and clients. Moreover, we

have discovered that the rekeying protocol is vulnerable to a theoretical *unknown key-share* (UKS) attack (Blake-Wilson and Menezes, 1999): a malicious client *B*, with the help of another client *E*, can induce a client *A* to believe that she (still) shares a secret key with *E*, and instead *A* shares the key with *B*. The practical exploitability of this attack in actual implementations is still to be investigated. Our formalization covers also the behaviour of the users, when relevant; e.g., if the users do not check the fingerprints of their shared keys, a MitM attack is possible.

Vulnerabilities may arise also 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 aspects deserve further investigation and particular care in the implementation and use of this protocol.

In this work, the basic encryption primitive of MTProto 2.0 has been modeled as a perfect authenticated encryption scheme. Although no attack on this scheme is known to date, in order to deem MTProto 2.0 secure we need to prove properties such as IND-CCA and INT-CTXT. 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 (Blanchet, 2007; Barthe et al., 2013)—which we leave to future work. However, even if a flaw were found in the current encryption scheme, the results in this paper would still be valid: the encryption scheme could be replaced with a stronger one, and no other changes would be required.

Besides automatic tools like ProVerif and CryptoVerif, one may analyse (cryptographic) protocols in semi-automatic, interactive tools. A prominent example is EasyCrypt mentioned above, but there are also formalisations of the applied π-calculus or even general temporal logics in proof assistants like Coq or Isabelle/HOL (see, e.g. (Kahsai and Miculan, 2008; Miculan, 2001; Honsell and Miculan, 1995)). In principle, these interactive tools allow us to formalize and prove any properties which can be proved "on the paper", but the burden on the user is greater than that induced by automatic tools.

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 (Bhargavan et al., 2010; Pozza et al., 2004). Also, particular attention must be paid to side-channel attacks, such as on timing or traffic analysis. Another issue concerning the correct implementation of clients is that a server can craft malicious DH parameters, e.g., choosing generators that make discrete logarithms

significantly easier to compute (Kobeissi et al., 2017) 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 (Agrawal et al., 2004; Lenstra Jr and Pomerance, 2019).

Correct user behaviour is crucial in order to prevent MitM attacks in secret chats. As we have seen, 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. In practice, such checks are likely often ignored, or performed through the very same (supposedly secure) chat. Hence, users seriously concerned about privacy must be educated about the correct procedure to follow.

REFERENCES

Agrawal, M., Kayal, N., and Saxena, N. (2004). Primes is in p. *Annals of mathematics*, pages 781–793.

Barthe, G., Dupressoir, F., Grégoire, B., Kunz, C., Schmidt, B., and Strub, P.-Y. (2013). Easycrypt: A tutorial. In *Foundations of security analysis and design VII*, pages 146–166. Springer.

Bhargavan, K., Blanchet, B., and Kobeissi, N. (2017). Verified models and reference implementations for the TLS 1.3 standard candidate. In *IEEE European Symposium on Security and Privacy*, pages 483–502.

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

Blake-Wilson, S. and Menezes, A. (1999). Unknown key-share attacks on the Station-to-Station (STS) protocol. In *International Workshop on Public Key Cryptography*, pages 154–170. Springer.

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

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

Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., and Stebila, D. (2017). A formal security analysis of the Signal messaging protocol. In *IEEE European Symposium on Security and Privacy*, pages 451–466.

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

Frosch, T., Mainka, C., Bader, C., Bergsma, F., Schwenk, J., and Holz, T. (2016). How secure is textsecure?

In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 457–472. IEEE.

Honsell, F. and Miculan, M. (1995). A natural deduction approach to dynamic logic. In Berardi, S. and Coppo, M., editors, *Types for Proofs and Programs, International Workshop TYPES'95, Selected Papers*, volume 1158 of *Lecture Notes in Computer Science*, pages 165–182. Springer.

Jakobsen, J. (2015). A practical cryptanalysis of the Telegram messaging protocol. Master's thesis, Aarhus University.

Jakobsen, J. and Orlandi, C. (2016). 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.

Kahsai, T. and Miculan, M. (2008). Implementing spi calculus using nominal techniques. In Beckmann, A., Dimitracopoulos, C., and Löwe, B., editors, *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings*, volume 5028 of *Lecture Notes in Computer Science*, pages 294–305. Springer.

Kobeissi, N., Bhargavan, K., and Blanchet, B. (2017). 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. IEEE.

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

Miculan, M. (2001). On the formalization of the modal μ-calculus in the calculus of inductive constructions. *Inf. Comput.*, 164(1):199–231.

Pozza, D., Sisto, R., and Durante, L. (2004). Spi2java: Automatic cryptographic protocol Java code generation from spi-calculus. In *18th International Conference on Advanced Information Networking and Applications. AINA 2004.*, volume 1, pages 400–405. IEEE.

Rad, A. and Rizzo, J. (2015). A 2^{64} attack on Telegram, and why a super villain doesn't need it to read your telegram chats. http://www.alexrad.me. Accessible via https://web.archive.org.

Rösler, P., Mainka, C., and Schwenk, J. (2018). More is less: on the end-to-end security of group chats in Signal, WhatsApp, and Threema. In *IEEE European Symposium on Security and Privacy*, pages 415–429.

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

Telegram (2021a). MTProto mobile protocol. https://core.telegram.org/mtproto/ (last accessed on Febuary 15, 2021).

Telegram (2021b). Telegram FAQ for the Technically Inclined. https://core.telegram.org/techfaq (last accessed on February 15, 2021).