UNIVERSITY OF UDINE
hic sunt futura

Department of
Mathematics, Computer Science and Physics

BACHELOR THESIS IN
INTERNET OF THINGS, BIG DATA AND WEB

Symbolic Verification of MTProto 2.0: a comparative study

CANDIDATE
Alessandro Zanatta

SUPERVISOR
Prof. Marino Miculan

CO-SUPERVISOR
Prof. Nicola Vitacolonna

Academic Year 2020-2021

Institute Contacts
Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine — Italia
+39 0432 558400
https://www.dmif.uniud.it/

Acknowledgements

Throughout the writing of this thesis, I have received the support and assistance of many people.

First of all, I would like to thank my supervisor, Professor Marino Miculan, and my co-supervisor, Professor Nicola Vitacolonna. Their expertise, knowledge, and advice were invaluable to concluding with positive results this research.

I also want to thank every professor of the Internet of Things, Big Data and Web degree. They all have contributed to my personal and academic growth and I am grateful to each one.

I would like to thank my parents, which helped me during these three years, both economically and emotionally. They have always backed my every decision, ever since I chose my course of study.

A sincere thank you to my friends and colleagues, Christian Abbondo and Fabiana Rapicavoli, with whom I have shared my studies for more than two years. Thank you for making my university life more vibrant, cheerful, and delightful.

A special thanks to Letizia Polla, which has always been by my side when I most needed support, especially during the harsh months of the pandemic. I could not have completed this thesis without her patient backing.

I want to dedicate this milestone to all the people I have mentioned so far.

Contents

1

Introduction

Security protocols are used every day by billions of users and applications to guarantee a certain degree of security and privacy over communications happening on the (insecure) Internet (SSH [39] and TLSv1.3 [26] protocols are just a few famous examples). Although, there is a catch: designing such protocols has been proven to be very error-prone. As an example, consider the Needham-Schroeder public-key protocol [48], which has been believed to be secure for almost 20 years - before a fatal flaw was found and corrected [40].

Given the importance of the correctness of such protocols and the difficulties for designers to ensure it, it has been necessary to *formally* prove the absence of security vulnerabilities. Towards this aim, a set of tools has been developed to assist the designing of a new protocol.

In this thesis, we are going to compare two tools: Tamarin prover and Proverif. In particular, we will compare them for the formal verification of Telegram's protocol: MTProto2.0. The protocol has already been analyzed and formalized with Proverif by M. Miculan and N. Vitacolonna [42], and it has been discussed in the associated paper [44]. The formalization in Tamarin will be based on this work, with remarkable differences due to the diverse nature of the two tools.

In chapter 2, we will present the symbolic and computational models. The two tools, Tamarin prover and Proverif, will be briefly described in chapter 3. In the next chapter, we will also present the model of a very simple protocol with both tools. A description of the MTProto2.0 protocol will be given in chapter 5, while, in chapter 6, we will describe its formalization in Tamarin prover. Finally, in chapter 7, we will compare Tamarin prover and Proverif.

2

Symbolic and computational model

First of all, let us consider the different types of approaches to security protocol analysis. The two categories of techniques are shown in fig. 2.1. We will proceed to examine the symbolic and computational models in this chapter.

In the *symbolic model* (often called Dolev-Yao model) [22], the cryptographic primitives are considered as black-box and are represented using function symbols, the messages are terms, and the adversary can only use defined primitives. An important aspect to note of this model is that it assumes **perfect cryptography**. As an example, consider the case in which there are two function symbols (**enc** and **dec**, used to encrypt and decrypt), a message m and a key k and the following equality is defined:

$$\mathrm{dec}\left(\mathrm{enc}\left(m,k\right),k\right)=m \tag{2.1}$$

Following from the equation − and considering the perfect cryptography assumption − it is possible to decrypt $\mathrm{enc}\left(m,k\right)$ if and only if k is known [12].

In the *computational model* the messages are bitstrings, the cryptographic primitives are functions from bitstrings to bitstrings, and the attacker is modeled as a probabilistic Turing machine. A security property in this model is considered to hold when the probability that it does *not* hold is negligible. For instance, the previously discussed shared-key encryption can be modeled using the same equation. However, the security of encryption is expressed by stating that the attacker has an insignificant probability of breaking the primitive (e.g. decrypting the message without having the key). Security proofs using this model are usually stronger, but this comes to the cost of long, complex, tedious, and highly error-prone proofs (as stated by INRIA researchers [17]). Finally, as pointed to by B. Blanchet [12], the computational model is indeed just a *model* and ignores many aspects of reality, and potential attacks, for example, faulty attacks like the one affecting processors computing RSA signatures [46].

Of all the tools in fig. 2.1, we are going to discuss two automatic tools that employ a symbolic model: Proverif [15] and Tamarin prover [20]. We will also refer to Tamarin prover as Tamarin for brevity.

We focus on the tools that exploit the symbolic model as these make it possible to automate proofs. Termination is still not always guaranteed[1]. From now on, we will refer to the symbolic model implicitly.

[1]It depends on the tool. Both Proverif and Tamarin may not terminate. Proverif proofs may end with an inconclusive result, while Tamarin will always give the correct answer (assuming termination). Other tools, like Scyther [18], always terminate by limiting the growth of some parameters. More details about this problem will be discussed in section 2.1.

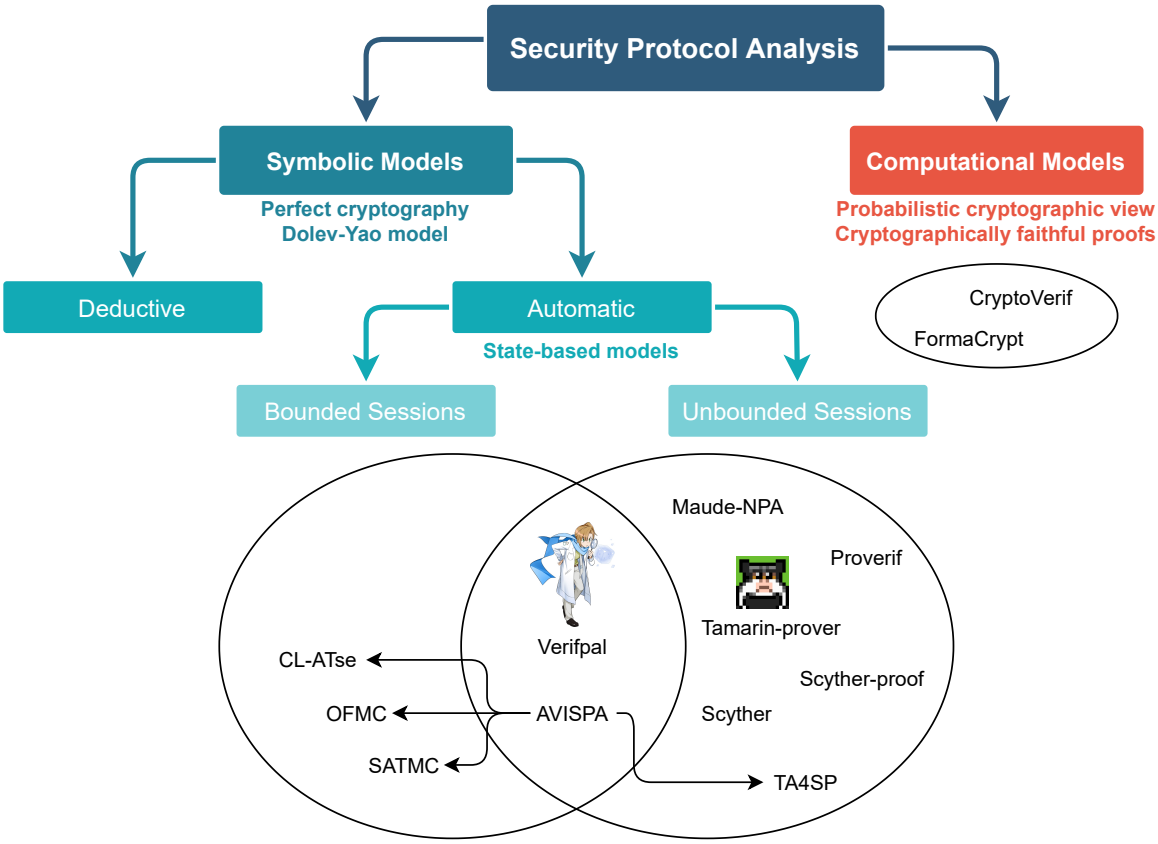


Figure 2.1: Symbolic and computational models and available tools.
Inspired by a representation from Nicola Vitacolonna.

2.1 Difficulties in the security protocol analysis in the symbolic model

Two main problems affect most automatic tools for security protocol analysis: **infinite state space** and **undecidability**.

At a high level, to verify, for example, secrecy in the symbolic model, one computes the set of terms that the adversary knows. If a certain term does not belong to this set, then it is considered secret [13]. The difficulty is that the set of terms is infinite. Specifically, we have to deal with three sources of infinity when analyzing a protocol:

Messages

the adversary can produce messages of any arbitrary size;

Sessions

as many attacks are possible only when multiple sessions are executed in parallel, symbolic models often use an unbounded number of sessions;

Nonces

if we have an unbounded number of session, then we also must have an unbounded number of nonces to use in those sessions.

There have been various ways of tackling this problem:

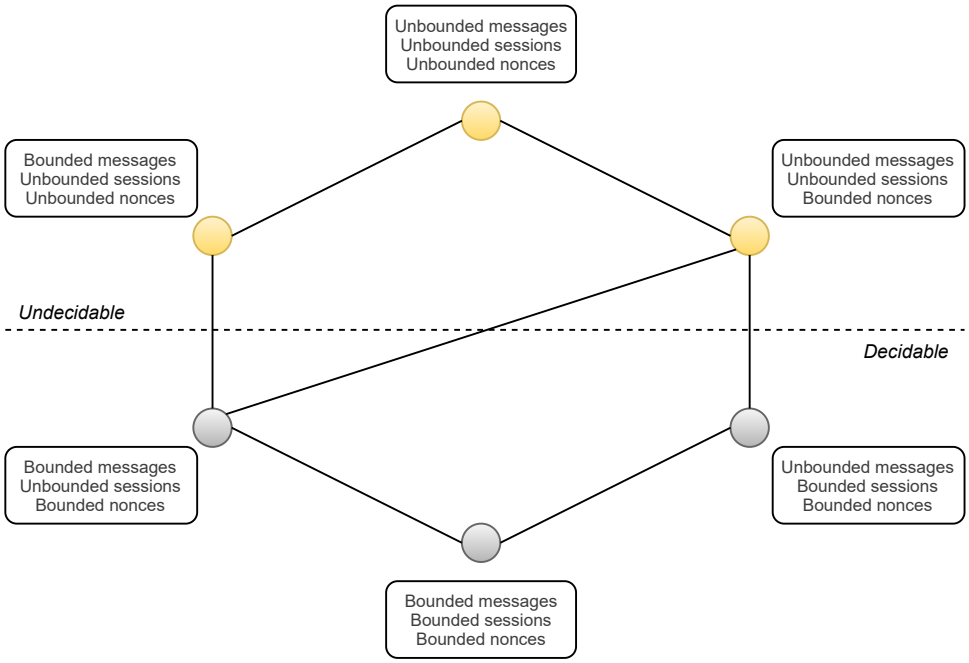


Figure 2.2: Decidability of termination.
Inspired by a representation from Nicola Vitacolonna.

- It is possible to decide to bound every source of infinity. In this case, the state space is finite. This method was used, for example, by Lowe [41] and SATMC [8];
- We can bound the number of executions of the protocol. This still leads to an infinite state space, but it has been shown that insecurity is NP-complete [49];
- If we do not force any bound, the problem becomes undecidable [25] [28]. As pointed out by B. Blanchet [13], no automatic tool that always terminates and solves the problem can exist. In general, it is not possible to determine if a certain proof will, eventually, end. Figure 2.2 shows the boundary of decidability. As evidenced by the diagram, the problem is decidable if and only if we bound at least two sources of infinity out of three.

  Of course, there are several approaches to confront this problem:

  – Rely on help from the user. This is the approach chosen by Tamarin [50], which allows for semi-automatic proofs. In Tamarin, possible resolution steps are ranked by the system, but the user can choose which to solve first;
  – Another possibility is to have tools that may return an inconclusive result − yet are supposed to work correctly in most cases. Proverif follows this approach [13];
  – The last approach is to allow non-termination.

3

Tools foundations

In this section, we will look at the foundations of the analyzed tools, Proverif and Tamarin prover.

3.1 Proverif

Let us start with a brief overview of Proverif internal reasoning. For more information, please refer to [12, 13, 14].

3.1.1 High level view

Proverif protocol specification is based on an extended version of the π-calculus (the *applied* π-calculus). The tool also allows the user to define constructors, destructors and equations[1], which form the cryptographic primitives. Security properties are modeled using Horn clauses. The entire protocol is then automatically translated to a set of Horn clauses. Using this abstract representation of the protocol, the Proverif verifier uses a resolution algorithm on such clauses that allow for verification of security properties [12]. A graphical representation of the process is given in fig. 3.1.

It is important to note that Proverif is not complete. We will consider implications in chapter 7. Moreover, it may not terminate, but it has been proven to be precise and efficient enough in practice by many case studies (the following is a non-exhaustive list of examples [1, 4, 44, 47]). We will now proceed with an overview of π-calculus and Horn clauses.

3.1.2 π-calculus and applied π-calculus

The π-calculus [45] is a (minimal) programming language that models a system communicating on channels. It belongs to the *process calculi* family, which is generally used to model concurrent systems. As Proverif uses the *applied* π-calculus (which is an extension of standard π-calculus), we will briefly present its syntax in the rest of this section.

The following description of applied π-calculus references articles [2, 3, 29]. Please refer to these resources for further information and a more formal or in-depth description. For brevity, we only define the main features of applied π-calculus in section 3.1.2.

[1] Destructors are basically used to de-construct some previously constructed term (e.g. decryption of an encrypted ciphertext), while equations represent term equality of some sort (e.g. commutativity of multiplication).

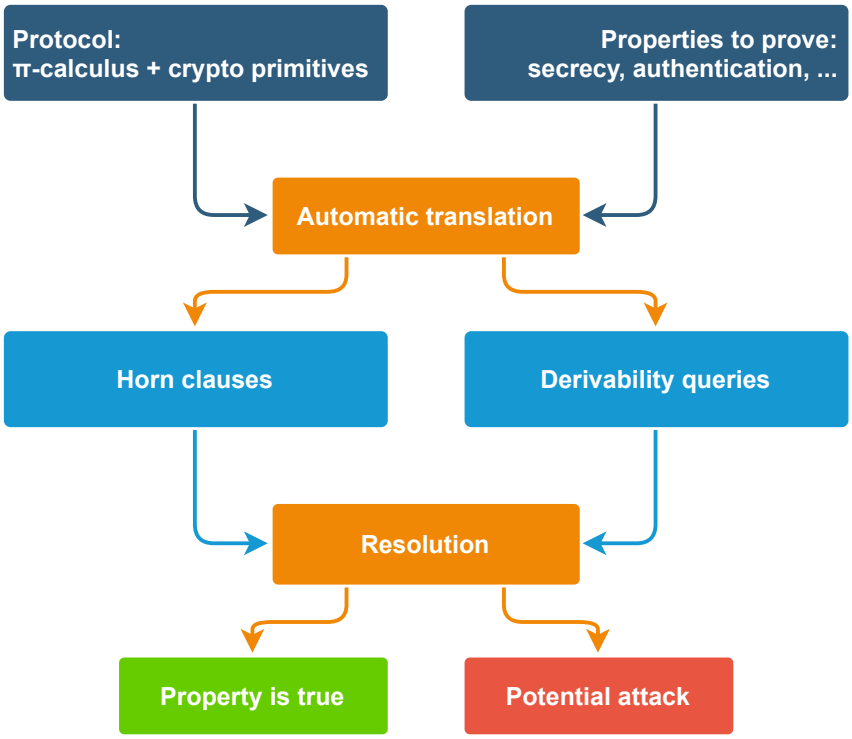


Figure 3.1: Proverif verification method.
Inspired by a representation from B. Blanchet [12].

Overview of the syntax of the applied π-calculus A *signature* Σ is composed by a finite number of functions symbols, each with its own integer arity. Given such signature, together with an infinite set of names and an infinite set of variables, the set of **terms** is defined by the grammar:

$$\begin{array}{lll} U, V ::= & & \text{terms} \\ & a, b, \ldots & \text{name} \\ & x, y, \ldots & \text{variable} \\ & \mathrm{f}(U_1, \ldots, U_l) & \text{constructor application} \end{array} \tag{3.1}$$

where $f \in \Sigma$ and l matches the arity of f. Next, we define a grammar for processes, which is shown in eq. (3.2). As pointed to by Microsoft researchers, this grammar is very similar to the π-calculus [29]. We will omit to define differences from standard π-calculus as we have not formally defined π-calculus either.

$$\begin{array}{lll} P, Q ::= & & \text{processes} \\ & 0 & \text{null process} \\ & c(x).P & \text{message input} \\ & \overline{c}\langle x \rangle.P & \text{message output} \\ & P \mid Q & \text{parallel composition} \\ & !P & \text{replication} \\ & \nu n.P & \text{name restriction (“fresh”)} \\ & \text{if } M = N \text{ then } P \text{ else } Q & \text{conditional} \end{array} \tag{3.2}$$

The null process 0 does nothing; $c(x).P$ ($\overline{c}\langle x\rangle.P$) gets (outputs) the message x from (into) channel $\overline{c}$ (c) and then continues with process P; Notice that getting a message from a channel is a blocking operation; $P \mid Q$ is the parallel composition of P and Q; The process $!P$ effectively behaves as an infinite number of copies of P running in parallel (*unbounded* replication); $vn.P$ creates a new fresh value, before proceeding with process P; if M then P else Q is a standard conditional.

3.1.3 Horn clauses

As we said earlier, Proverif's analysis is based on Horn clauses. Let us define them.

The class of Horn formulas is obtained by restricting the form of the conjuncts in a proposition in conjunctive normal form. Consider a proposition A in conjunctive normal form $C_1 \wedge \ldots \wedge C_n$, where each C_i is a conjunction of either positive or negative literals. A is a Horn clause if and only if each C_i contains at most one positive literal [23, 34]. Additionally, every variable in a clause is implicitly universally quantified.

3.1.4 Intuition for the translation to Horn clauses

For this section we will consider Horn clauses in the following form: $F_1 \wedge \ldots \wedge F_n \Rightarrow F$. Additionally, the literal $\text{attacker}(x)$ will be used to denote that the attacker knows the term x.

Proverif automatically translates cryptographic primitives, attacker capabilities, and the protocol itself to Horn clauses. In the following paragraphs, we will show some translations that Proverif applies. Most of the examples are taken from B. Blanchet [11].

Representation of cryptographic primitives As already seen, Proverif represents cryptographic primitives as constructors and destructors. A destructors g is defined as a set $\text{def}(g)$ of rewrite rules of the form $g(M_1, \ldots, M_n) \to M$, where $M_1, \ldots, M_n, M$ are terms that contain variables and constructors and the variables in M all occur in $M_1, \ldots, M_n$.

Representation of attacker capabilities During its computation, the attacker can apply constructors and destructors. Let us define a constructor of arity n as $\mathrm{f}(x_1, \ldots, x_n)$. We can model it with the following clause:

$$\text{attacker}(x_1) \wedge \ldots \wedge \text{attacker}(x_n) \Rightarrow \text{attacker}(\mathrm{f}(x_1, \ldots, x_n)) \tag{3.3}$$

This is consistent with the symbolic model: the adversary needs to know every term to apply a constructor. Now, suppose that g is a destructor. For each rewrite rule $\mathrm{g}(M_1 \wedge \ldots \wedge M_n) \to M$ in $\text{def}(g)$ we have the following clause:

$$\text{attacker}(M_1) \wedge \ldots \wedge \text{attacker}(M_n) \Rightarrow \text{attacker}(M) \tag{3.4}$$

Notice that the destructor did not appear in the Horn clause, and never will. Instead, we use pattern matching to model them. For example, let us show how we can model public-key encryption:

$$\begin{array}{ll} \text{aenc} & \text{attacker}(m) \wedge \text{attacker}(pk) \Rightarrow \text{attacker}(\text{aenc}(m, pk)) \\ \text{pk} & \text{attacker}(sk) \Rightarrow \text{attacker}(\text{pk}(sk)) \\ \text{adec} & \text{attacker}(\text{aenc}(m, \text{pk}(sk))) \wedge \text{attacker}(sk) \Rightarrow \text{attacker}(m) \end{array} \tag{3.5}$$

The first two equation model the constructors $\text{aenc}(m, pk)$ and $\text{pk}(sk)$. The third models asymmetric decryption, defined as $\text{adec}(\text{aenc}(m, \text{pk}(sk)), sk)$.

We also model a revealing signing scheme, formed by two constructors $\text{sign}(m, sk)$ and $\text{pk}(sk)$ and from the rewrite rules $\text{getmess}(\text{sign}(m, sk)) \rightarrow m$ and $\text{verify}(\text{sign}(m, sk), \text{pk}(sk)) \rightarrow m$, respectively used for message revealing and for signature verification. Equation (3.6) shows the representation of this primitives as Horn clauses.

$$\begin{array}{ll} \text{sign} & \text{attacker}(m) \wedge \text{attacker}(sk) \Rightarrow \text{attacker}(\text{sign}(m, sk)) \\ \text{getmess} & \text{attacker}(\text{sign}(m, sk)) \Rightarrow \text{attacker}(m) \\ \text{verify} & \text{attacker}(\text{sign}(m, sk)) \wedge \text{attacker}(\text{pk}(sk)) \Rightarrow \text{attacker}(m) \end{array} \tag{3.6}$$

Representation of the protocol Let us show how the protocol below can be written as Horn clauses. We additionally leave the attacker the task of starting the protocol, that is, the attacker will send the public key of the responder to A.

1. A $\rightarrow$ B: $\text{aenc}(\text{sign}(k, skA), pkB)$
2. B $\rightarrow$ A: $\text{senc}(m, k)$

The responder of the protocol can either be B or the attacker. In both cases, we can model this with:

$$\text{attacker}(\text{pk}(x)) \Rightarrow \text{attacker}(\text{aenc}(\text{sign}(k, skA), \text{pk}(x))) \tag{3.7}$$

When B receives the message, he decrypts it with his secret key skB. Then, B tests the signature evaluating $\text{verify}(x', pkA)$, which succeeds if and only if $x' = \text{sign}(y, skA)$. If so, B sends a secret constant m using the encryption key k. We assume that the attacker relays the message coming from A, and intercepts the message sent by B. Hence, the following clause models the second message of the protocol:

$$\text{attacker}(\text{enc}(\text{sign}(y, skA), \text{pk}(skB))) \Rightarrow \text{attacker}(\text{senc}(m, y)) \tag{3.8}$$

3.2 Tamarin prover

In this section, we will see an overview of Tamarin's foundations and internal reasoning. For a more in-depth description and further information, see the Tamarin foundations paper [50] or the extended foundations paper [51].

3.2.1 High level view

First of all, let us examine a high-level picture of Tamarin.

The security property model of Tamarin is based on labeled multiset rewriting rules to specify protocols and adversary capabilities, a guarded fragment[2] of first-order logic to specify security properties[3] and functions and equational theories to model the algebraic properties of cryptographic protocols [50]. Finally, Tamarin uses a novel constraint-solving algorithm to validate or falsify lemmas.

In other words, Tamarin allows to specify a labeled transition system that induces a set of traces and offers automatic verification of such traces using a guarded fragment of first-order logic to specify "good" traces. Tamarin then tries to prove the negation of the specified "good" traces.

Tamarin also offers built-in equational theories [52]. A brief overview will be given in section 3.2.4.

3.2.2 Terminology

As reported earlier, multiset rewriting rules are used to specify adversary capabilities and protocols. More precisely, a *set* of *labeled* multiset rewriting rules are used.

The ingredients of this multiset rewriting system are the following:

Terms

which can be essentially thought of as messages. Terms can be of three different sorts. The more general sort is the *msg* sort, which has two incomparable sub-sorts *fresh* and *pub* for fresh and public names, respectively;

Facts

which model information in the protocol. Facts have an arity, can be linear or persistent and are composed by terms. Linear facts model resources that can be consumed once, while persistent facts can be consumed an arbitrary number of times (and are prefixed by an exclamation mark). By convention, facts always start with a capital letter;

Special facts

Four facts are reserved and are used to model the freshness of a message t ($\mathbf{Fr}(t)$), a message t coming from the public channel ($\mathbf{In}(t)$), a message t to send to the public channel ($\mathbf{Out}(t)$) and knowledge of a certain message t from the attacker ($\mathbf{K}(t)$);

State of the system

The state of the system is represented using a *multiset* of facts;

Transition rules

A multiset of transition rules defines the possible transitions from one state to another one. Transitions are denoted with the following syntax

$$[L] -[A]\rightarrow [R] \tag{3.9}$$

where L, A and R are multisets of facts, respectively called **premises**, **actions** and **conclusions**.

[2] Only a few examples of formulas respecting the guarded fragment of first-order logic used by Tamarin will be given in section 3.2.5. See [6] for a rigorous definition from a mathematical point of view.

[3] Security properties in Tamarin will also be referred to as *lemmas*.

Trace

A trace is a sequence $\langle A_1, \ldots, A_n \rangle$ of sets of ground facts denoting the sequence of actions that happened during a protocol's execution.

3.2.3 Transition rules

Let us examine an informal description of transitions.

- Let S be the current state of the system
- Let $[L] \rightarrow [R]$ be a transition rule. Note that this is a *multiset rewriting rule* without a *label*;
- Let $[l] \rightarrow [r]$ be a ground instance of the rule (i.e. no variables are found in the multisets);
- If we apply $[l] \rightarrow [r]$ (assuming $l \subseteq^{\#} S$) to our state S we reach a new state, defined by the following equation:

$$S' = S \setminus^{\#} l \cup^{\#} r \tag{3.10}$$

  We use $\setminus^{\#}$, $\cup^{\#}$ and $\subseteq^{\#}$ to define difference, union and subset over multisets, respectively. We can also consider the difference between linear and persistent facts and define $lin\,(l)$ ($pers\,(l)$) as linear facts (persistent facts) in l. Assuming that $lin\,(l) \subseteq^{\#} S$ and $pers\,(l) \subseteq^{\#} S$, then the equation becomes the following:

$$S' = S \setminus^{\#} lin\,(l) \cup^{\#} r \tag{3.11}$$

  It should be clear from the equation why persistent facts can be consumed any number of times: they are never removed from the state of the system. This can be useful in scenarios in which we want to model persistent knowledge. For example, the establishment of an encryption key k may be expressed by a persistent fact $!\mathrm{Key}\,(k)$.

- When we use labeled multiset rewriting rules, such as $[l] -[a]\rightarrow [r]$, we also add facts from a to the *trace* of the execution.

Equation (3.12) shows multiset rewriting rules that are always defined by Tamarin. It is not hard to see that these equations are used to model the Dolev-Yao attacker: the first rule allows the adversary to send a message x he knows on the channel, while the second one allows him to learn a message x sent by someone. Notice that knowledge is, of course, persistent.

$$\begin{gathered} [!KU\,(x)] -[K\,(x)]\rightarrow [In\,(x)] \\ [Out\,(x)] \rightarrow [\,!KD(x)] \end{gathered} \tag{3.12}$$

3.2.4 Built-in equational theories

A brief list of Tamarin built-ins is given below. Only the built-in theories considered relevant and those used in the analysis will be described here. The complete list is available in the Tamarin manual [52].

hashing

defines a perfect hash function $\mathbf{h/1}$[4];

[4]The writing $\mathbf{f/x}$ indicates that the function $\mathbf{f}$ has arity $\mathbf{x}$.

symmetric-encryption

models a symmetric encryption scheme. It defines two symbols: **senc/2** and **sdec/2**, joined by the equation **sdec(senc(msg, k), k) = msg**;

asymmetric-encryption

models a public key encryption scheme. It defines the following symbols:

- **aenc/2**, used to model the encryption of a message with a public key
- **adec/2**, used to model the decryption of an encrypted message with a private key
- **pk/1**, used to derive a public key from a private key

Functions are related by the equation **adec(aenc(msg, pk(sk)), sk) = msg**;

diffie-hellman

models Diffie-Hellman groups. It defines the following symbols:

- **inv/1**, models the inverse of an element
- **1/0**, models the neutral element
- ^ and ***** symbols, models exponentiation and multiplication respectively

The equational theory for this built-in is actually quite complex. For the sake of completeness, these are the related equations:

$$\begin{aligned} &x\hat{}y\hat{}z = x\hat{}\,(y * z) \\ &x^1 = x \\ &x * y = y * x \\ &(x * y) * z = x * (y * z) \\ &x * 1 = x \\ &x * inv\,(x) = 1 \end{aligned} \tag{3.13}$$

Notice that Tamarin is the only tool that models groups with such precision. Of course, having such complex equations might become a burden when considering efficiency and resources consumption.

xor and bilinear-pairing

the xor theory models the exclusive-or operation. It defines $\oplus$**/2** (also written as **XOR/2**) and **zero/0**. This theory faithfully models xor properties with the following equations:

$$\begin{aligned} &x \oplus y = y \oplus x \\ &(x \oplus y) \oplus z = x \oplus (y \oplus z) \\ &x \oplus zero = x \\ &x \oplus x = zero \end{aligned} \tag{3.14}$$

Additionally, the bilinear pairing theory extends the Diffie-Hellman theory, which allows working with elliptic curves.

These theories will not be used in the analysis. Nonetheless, they are noteworthy as there does not seem to be another tool that models them faithfully.

3.2.5 Guarded formulas

As reported earlier, Tamarin uses a guarded fragment of first-order logic to specify security properties and − in general − traces. All formulas must be guarded, which essentially means that all quantified variables **must** appear in facts. Equation (3.15) shows two main formulas that respect the guarded fragment. Most, if not every, security property can be expressed using these formulas:

$$\begin{aligned} \forall\, \overline{x}.\ F\,(\overline{z})\,@i \Rightarrow \psi \\ \exists\, \overline{x}.\ F\,(\overline{z})\,@i \wedge \psi \end{aligned} \tag{3.15}$$

where F is a fact, ψ is guarded and $\overline{x}$ and $\overline{z}$ are vectors of terms such that $\overline{x} \subseteq \overline{z} \cup i$. The variable i is a timepoint, which denotes that the fact F occurred at time i (timepoints all belong to $\mathbb{Q}$)[5] [19].

[5]Notice that timepoints are very useful to define perfect forward secrecy lemmas and, in general, post-compromise security properties (i.e. leaking an ephemeral key **after** honest parties have used it).

4

Modeling with Proverif and Tamarin prover

In this chapter, we will describe the usage of Proverif and Tamarin prover from a user perspective. We do so by showing the modeling process of the protocol shown in section 3.1.4. Again, we leave the attacker the task of starting the protocol, that is, the attacker will send the public key of the responder to the initiator.

4.1 Proverif

4.1.1 Cryptographic primitives

Proverif does not offer any built-in equational theory. Every time we want to use a cryptographic primitive, we need to define it ourselves.

In the example protocol, we will need symmetric and asymmetric encryption and revealing signing. First of all, as Proverif uses a typed language, we can create some types for the keys. We will use the built-in type **bitstring** for other terms.

```
1 type Key.
2 type PublicKey.
3 type PrivateKey.
```

Then, to define constructors, we simply define their signatures using types:

```
1 fun pk(PrivateKey): PublicKey.              (* Public key derivation from private key *)
2 fun aenc(bitstring, PublicKey): bitstring.  (* Asymmetric encryption *)
3 fun senc(bitstring, Key): bitstring.        (* Symmetric encryption *)
4 fun sign(bitstring, PrivateKey): bitstring. (* Signing *)
```

Finally, we define destructors:

```
1 (* Asymmetric decryption *)
2 reduc forall m: bitstring, sk: PrivateKey;
3     adec(aenc(m, pk(sk)), sk) = m.
4
5 (* Symmetric decryption *)
6 reduc forall m: bitstring, k: Key;
7     sdec(senc(m, k), k) = m.
```

```
8
9  (* Signature verification *)
10 reduc forall m: bitstring, sk: PrivateKey;
11     verify(sign(m, sk), m, pk(sk)) = true.
12
13 (* Signature message revealing *)
14 reduc forall m: bitstring, sk: PrivateKey;
15     getmess(sign(m, sk)) = m.
```

The syntax **reduc forall** $x_1 : t_1, \ldots, x_n : t_n;\ \mathrm{g}(M_1, \ldots, M_k) = M.$ is used to define a rewrite rule (with term types t_i). This declaration can be expended to define *a set* of rewrite rules or even *a sequence*. When a destructor fails, the process which executed it blocks. As already seen, every term on the right-hand side must appear on the left-hand side.

Proverif additionally allows the creation of type converters, which allow to cast a term from one type to another. We defined two type converters for the `Key` type:

```
1 fun key2bit(Key): bitstring [typeConverter].
2 fun bit2key(bitstring): Key [typeConverter].
```

Type converters are declared as constructors, with the additional `[typeConverter]` annotation.

4.1.2 Channels and events

We have seen in section 3.1.2 that the applied π-calculus is a language that models a system communicating on channels. In Proverif, we can create channels (both public or private).

```
1 channel io. (* Public channel used for every exchange *)
```

We also need to define events, which are used to define security properties. To declare an event, we need to define its name and its parameters types.

```
1 event AReceivedMessage(bitstring, Key).
2 event BSentMessage(bitstring, Key).
```

4.1.3 Protocol model

To define parties, we use the **let** construct, which allows creating a parameterized process macro.

Let us start by defining the process executed by the initiator:

```
1  let A(skA: PrivateKey) =
2    in(io, pkX: PublicKey);
3
4    new k: Key;
5    let bit_k = key2bit(k) in
6    out(io, aenc(sign(bit_k, skA), pkX));
7
8    in(io, c: bitstring);
9    let m = sdec(c, k) in
10       event AReceivedMessage(m, k);
11   0.
```

On line 1, we define the macro `A` with a parameter `skA` (of type `PrivateKey`). Then, we receive the public key of the other party from the public channel. Notice that the **in**`(...)` operation is blocking (i.e. the process stops until it receives a message). On lines 4-6, we generate a new key, convert it to

a bitstring, and apply cryptographic functions before sending the message. Finally, on lines 8-10, we receive a message, and then we apply destructors using the `let-in` construct. If decryption is correct, we add the event `AReceivedMessage` to the trace.

Then, we define the responder process similarly:

```
1  let B(skB: PrivateKey, pkA: PublicKey) =
2    in(io, c: bitstring);
3    let signature = adec(c, skB) in
4
5    let bit_k = getmess(signature) in
6    if verify(signature, bit_k, pkA) then
7        let k = bit2key(bit_k) in
8        new m: bitstring;
9        out(io, senc(m, k));
10       event BSentMessage(m, k);
11   0.
```

On line 6, we use a conditional statement to check if the signature is correct. If it is, we create a new message, send it encrypted with key `k` and add the `BSentMessage` event to the trace.

Finally, we define the `process`, which defines which processes are executed. As we have parameterized our client processes, we additionally create their keys and pre-share them through arguments.

```
1  process
2    new skA: PrivateKey; let pkA = pk(skA) in out(io, pkA);
3    new skB: PrivateKey; let pkB = pk(skB) in out(io, pkB);
4    ((!A(skA)) | (!B(skB, pkA)))
```

Here we are executing an unbounded number of processes of both A and B in parallel. Notice that we output the public key of each principal to make sure the attacker has access to it.

4.1.4 Security properties

We define security properties as Horn clauses. For example, we can create a query for secrecy of the message in the following manner:

```
1  query m: bitstring, k: Key;
2    event(AReceivedMessage(m, k)) && event(BSentMessage(m, k)) && attacker(m) ==> false.
```

The `query` syntax requires to define terms type with the same syntax used before. In queries, we can use conjunctions (`&&`), disjunctions (`||`) implications (`==>`) and negation (`not`).

Proverif allows the usage of many other constructs when defining queries. For a complete list, please refer to the user manual [14].

4.2 Tamarin prover

4.2.1 Preamble and cryptographic primitives

First of all, we need to add the preamble of our Tamarin file. Tamarin files use the `.spthy` extension. In the opening, we specify the theory name with `theory ExampleProtocol` and import the built-in equational theories we need (`builtins: ...`). The protocol is enclosed between a `begin` and an `end` keyword.

```
1  theory ExampleProtocol
2  begin
```

```
3
4   builtins: asymmetric-encryption, symmetric-encryption, revealing-signing
5
6   /* Protocol model and security properties here */
7
8 end
```

4.2.2 Public Key Infrastructure definition

We start by defining a Public Key Infrastructure (PKI) that creates keypairs. We do it with the following rule:

```
1 rule CreateKeyPair:
2   let pkey = pk(~skey) in
3     [ Fr(~skey) ]
4   --[]->
5     [
6       !PublicKey($X, pkey),
7       !PrivateKey($X, ~skey),
8       Out(pkey)
9     ]
```

There are many details to discuss:

- The meaning of the three square brackets is the same seen in section 3.2.2;
- The arrowed square bracket in the middle `--[]->` can also be written as `-->` if the multiset of action facts is empty;
- On line 2, we use the let-in construct. It allows us to create key-expression pairs and reuse them in the rule. Many pairs can be defined, each separated by a newline;
- We recall that `Fr(~skey)` models the creation of the fresh term `~skey`. In Tamarin, a tilde precedes fresh terms;
- Both `!PublicKey` and `!PrivateKey` facts are preceded by an exclamation mark, indicating they are persistent;
- Finally, terms preceded by a dollar sign belong to the public names sort.

We also give the attacker the ability to create keypairs with known private keys for a dishonest user:

```
1 rule CreateAttackerKeyPair:
2   let pkey = pk(~skey) in
3     [ In(X), Fr(~skey) ]
4   --[ Dishonest(X) ]->
5     [
6       !PublicKey(X, pkey),
7       !PrivateKey(X, ~skey),
8       Out(~skey)
9     ]
```

The action fact `Dishonest(X)` will be used to define security properties following our threat model.

4.2.3 Protocol model

Let us define the protocol rules. We will not describe these rules line by line as the inline comments should be sufficiently informative. Instead, we will give a general description of every rule.

```
1  rule A_1:
2    let
3      signed_key = revealSign(~k, skA) // Sign the key...
4      c = aenc(signed_key, pkX)        // ...and encrypt the signature
5    in
6      [
7        In(R),                   // Get responder identity...
8        !PublicKey(R, pkX),      // ...and use it to get its public key
9        !PrivateKey($A, skA),    // Get initiator private key
10
11       Fr(~k)                   // Generate fresh key
12     ]
13   -->
14     [
15       AState_1(pkX, skA, ~k), // Save state
16       Out(c)                  // Send the encrypted message
17     ]
18
19 rule B_2:
20   let
21     c = aenc(signed_key, pk(skB)) // Pattern match the incoming message
22     k = getMessage(signed_key)    // Recover key from signature
23   in
24     [
25       !PublicKey($A, pkA),  // Get public key of $A
26       !PrivateKey($B, skB), // Get private key of $B
27       In(c),                // Receive message
28
29       Fr(~m)                // Create fresh message to send to initiator
30     ]
31   --[
32       Eq(revealVerify(signed_key, k, pkA), true), // Verify signature
33       BSentMessage($B, $A, ~m, k)
34     ]->
35     [ Out(senc(~m, k)) ] // Send message ~m encrypted with key k
36
37 rule A_3:
38     [
39       AState_1(pkX, skA, k), // Get state from the first client rule
40       In(senc(m, k))         // Receive the message
41     ]
42   --[ AReceivedMessage($A, m, k) ]->
43     []
```

The first rule (`A_1`) models client A starting the protocol by sending the signed and encrypted key to the other party. As already seen in the Proverif model, we let the attacker choose the responder.

The second one (`B_2`) models the second part of the protocol: B gets its private key as well as A's public key, decrypts the incoming message, checks the signature, and sends the fresh encrypted message to A using the received key.

Finally, the last rule models A receiving the encrypted message.

Notice that the 2nd and 3rd rules have been labeled with two action facts. We will use these to model security properties.

4.2.4 Security properties

Finally, we define security properties (or *lemmas*). As already stated, lemmas use a guarded fragment of first-order logic section 3.2.5. Let us define a lemma to test the secrecy of the key. For every lemma, we will include comments that explain what the lemma models.

```
lemma Secrecy:
  "
    /* Whenever A and B exchange a message */
    ∀ A1 A2 B m k #i #j.
      AReceivedMessage(A1, m, k) @i ∧
      BSentMessage(B, A2, m, k) @j ∧

      /* and the initiator is honest */
      ¬(∃ #r. Dishonest(A1) @r) ∧

      /* and the responder answers only to the honest client A */
      ¬(∃ #r. Dishonest(A2) @r) ∧

      /* and is honest himself */
      ¬(∃ #r. Dishonest(B) @r)
      ⟹
      /* then the attacker does not know the message */
      ¬(∃ #r. K(m) @r)
  "
```

The first-order logic formula used should be easily readable with the support of comments. We introduce the last type of term: temporal values (`#i #j`), which are used as timepoints in lemmas. The syntax `Fact(term)@i` assigns the instant the fact `Fact(term)` was created to the timepoint `#i`.

Tamarin additionally allows creating lemmas that attempt to find a trace containing given facts. This feature can be useful to debug and test the protocol.

```
lemma ProtocolCanBeExecuted:
  exists-trace
  "∃ S k #i. AcceptKey(S, k) @i"
```

4.2.5 Restrictions

Tamarin supports restrictions on the trace. Informally, restrictions specify constraints that a protocol execution should uphold. In the rule `B_2` we have used the action fact `Eq`, which never appears in lemmas. As it is, the `Eq` does nothing more than an ordinary action fact. We then add the restriction, which uses the same guarded fragment of first-order logic employed by lemmas:

```
restriction RestrictionEqual:
  "∀ X Y #i. Eq(X, Y) @i ⟹X = Y"
```

In this example, we used the restriction to verify the correctness of the signature.

4.2.6 Partial deconstructions

When modeling, partial deconstructions are a recurrent problem. In the precomputation phase, Tamarin goes through all rules and inspects their premises. For each of these facts, Tamarin will precompute a set of possible sources, that is, the combinations of rules from which a fact can be obtained. However, sometimes Tamarin cannot resolve where a fact has come from [52].

The existence of such partial deconstructions complicates automated proof generation and often means that no proof will be found automatically (in reasonable time). To fix this issue, Tamarin allows defining *source lemmas*. Sources lemmas are a special type of lemmas and are applied during Tamarin's precomputation. Tamarin's manual also contains some tips on how to avoid partial deconstructions in the first place, for example:

- If the deconstruction happens on a term `t` that could be public, add a `In(t)` fact in the premises;
- Assign strict types: if a term should always be atomic (fresh or public) but pre-computation try to derive non-atomic terms, then give them the corresponding type;
- Use pattern matching instead of destructor functions;
- Start with a simple, working model, then refine it.

For an in-depth analysis of source lemmas (also called type assertions), please refer to [43].

5

MTProto2.0 protocol description

In this chapter, we will give a brief overview of the MTProto2.0 protocol. A more in-depth and formal description can be found on the official web page [55].

First of all, MTProto2.0 is a *suite of protocols* used to enable secure communication between a Telegram client and a Telegram server over an insecure network. MTProto2.0 can be decomposed in the three following protocols:

Authorization
: used to obtain a secret authorization key shared only with the server;

Secret chat
: used to obtain a secret key shared between two clients. This is then used to exchange end-to-end messages between two clients, with the server that basically acts as a forwarder;

Rekeying
: used to achieve perfect forward secrecy, allows obtaining a new end-to-end encryption key.

Additionally, the cloud chats encryption schema is used to securely exchange messages between clients and servers (who have shared an authorization key).

An overview on these protocols will be given in sections 5.1 to 5.4.

5.1 Authorization protocol

The first time a Telegram client C runs the application, it must negotiate an **authorization key** with the Telegram server S. The authorization protocol is used to this end. Once the client and the server have shared an authorization key, they will use it to encrypt (almost) every future communication between them. A client might also have several keys (e.g. on multiple devices or if reinstalling the application), some of which might be locked (e.g. if the device is lost). The authorization protocol is based on the Diffie-Hellman key exchange protocol [21].

A successful protocol run consists of three rounds, which are represented schematically in fig. 5.1:

Round 1
: In the first round, messages are in plaintext. In particular, the client and the server exchange

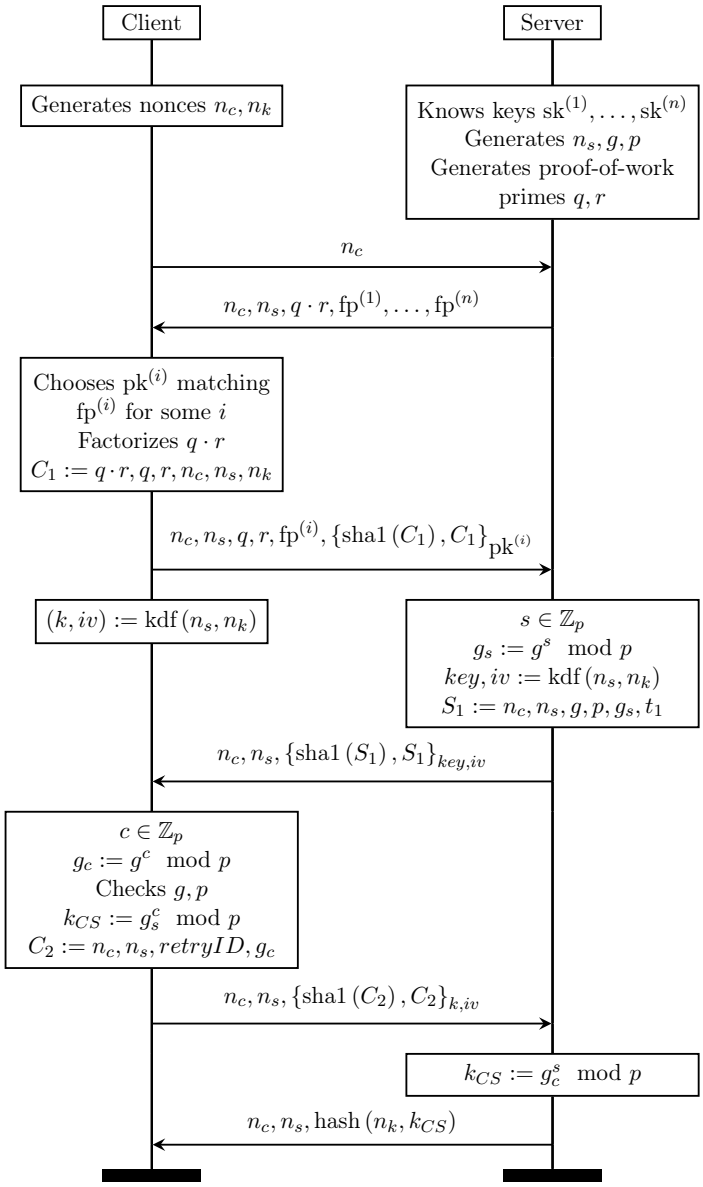


Figure 5.1: MTProto2.0 Authorization protocol

two nonces (n_c and n_s). The pair $\langle n_c, n_s \rangle$ identifies a session of the authorization protocol. These nonces are sent in every consequent message of the current run of the protocol, both in plaintext and encrypted form.

1. In the first message, the client sends his fresh nonce n_c to the server;
2. The server answers with the client nonce n_c, the server fresh nonce n_s, a challenge $q \cdot r \leq 2^{63}$ (which are two primes used as a measure to prevent denial of service, as the client needs to spend resources on factorizing $q \cdot r$ before the server has to commit (more) resources[1]) and a list of public RSA key fingerprints (calculated as the lower 64-bits of the SHA1 of the server public keys).

[1]Notice that this might not be true as this might be vulnerable to a lookup table approach (e.g. using factordb.com).

Round 2

The client decomposes $q \cdot r$ in $\langle q, r \rangle$ and retrieves the public key of the server $pk^{(i)}$. The client then generates a nonce n_k of 256 bits. This nonce n_k is supposed to be secret. The pair $\langle n_s, n_k \rangle$ is used, by both server and client, to derive, through a derivation function kdf, a symmetric encryption key k and an initialization vector iv, which will be used in subsequent exchanges.

3. The client asymmetrically encrypts both $\langle q \cdot r, q, r, n_c, n_s, n_k \rangle$ and its SHA1 hash with $pk^{(i)}$ and sends it along with $\left\langle n_c, n_s, q, r, fp^{(i)} \right\rangle$ to the server. A rather complex padding schema is used;
4. The server generates his Diffie-Hellman ephemeral key s of 2048-bits, chooses g, p and computes $g_s := g^s \mod p$. Finally, it symmetrically encrypts both $\langle n_c, n_s, g, p, g_s, t_1 \rangle$ and its SHA1 hash and sends it along with $\langle n_c, n_s \rangle$.

Notice that the client is supposed to check that:

- p is a safe 2048-bit prime, where safe means that both p and $\frac{p-1}{2}$ are prime and $2^{2047} < p < 2^{2048}$;
- g is a generator for $\frac{p-1}{2}$.

Round 3

In the last round, the client generates his own Diffie-Hellman ephemeral key and shares it with the server.

5. The client generates his ephemeral key c of 2048-bits and computes $g_c := g^c \mod p$. Then, it symmetrically encrypts both $\langle n_c, n_s, retryID, g_c \rangle$ and its SHA1 hash and sends them along with $\langle n_c, n_s \rangle$. The $retryID$ starts at zero at the time of the first attempt, otherwise is based on the values from the last failed attempt;
6. The server is now able to compute the authorization key as $k_{CS} := g_c^s \mod p$. Assuming server checks pass, S sends an acknowledgment for the new key: $\langle n_c, n_s, \text{hash}\,(k_{CS}) \rangle$.

5.2 Cloud chat encryption schema

Telegram uses the schema in fig. 5.2 to encrypt every message exchanged between the client and the server after an authorization key has been established using the authorization protocol in section 5.1.

A message key msg_key of 128 bits is calculated as the middle 128 bits of the SHA256 of the entire message prepended by 32 bytes of the authorization key. The message itself contains a 64-bit salt, a 64-bit session identifier, the payload[2] and a variable size padding of 12-1024 bytes. The authorization key auth_key, combined with the message key msg_key, is used to derive a key and an initialization vector, which are used to encrypt the entire message using AES in IGE mode.

[2]The payload contains the time of the message, its length, a sequence number. The receiver should check these pieces of information after decryption.

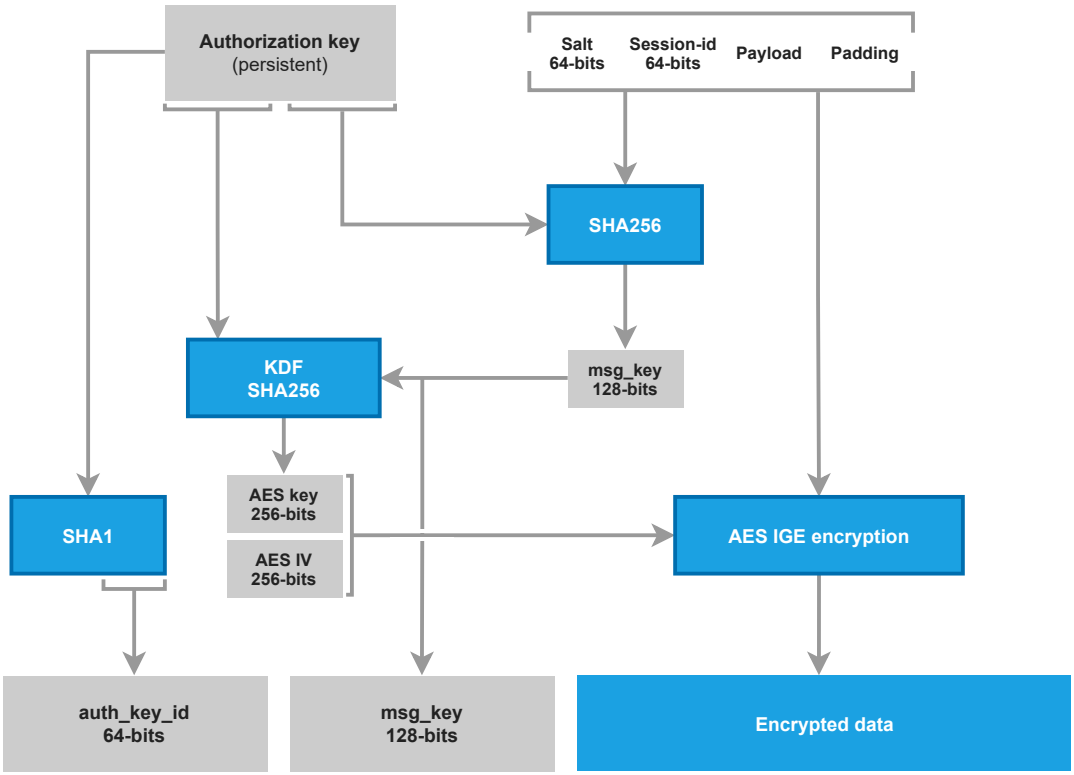


Figure 5.2: MTProto2.0 cloud chat protocol.
Representation inspired by the Telegram's official one.

5.2.1 IGE mode

Infinite Garble Extension (in short, IGE) is a block cipher mode, lesser-known than others like ECB, CBC, OFB, CTR, CFB, GCM, CCM. IGE can be defined with the following formula:

$$c_i = f_K(m_i \oplus c_{i-1}) \oplus m_{i-1} \tag{5.1}$$

where f_K stands for the encrypting function (like AES) with key K and i goes from 1 to n (the number of plaintext blocks). Two initialization vectors, c_0 and m_0, are also needed. Figure 5.3 summarizes how the encryption in IGE mode works.

One of the main properties of IGE mode is that it makes sure that if a ciphertext block is changed, then every subsequent block following it will not decrypt correctly.

As pointed out by [54], the Telegram developers team is aware of the vulnerability of this mode to blockwise-adaptive Chosen Plaintext Attack (CPA)[9], but they claim that MTProto2.0 is not affected.

5.3 Secret chat protocol

Telegram's secret chats deal with end-to-end encryption between two clients (as usual, let us call the clients Alice and Bob, or A and B for brevity). After both clients have shared an authorization key with the Telegram server S, they can decide to engage themselves in a run of the secret chat protocol, allowing them to share, using a Diffie-Hellman key exchange, a common secret. Notice that the server acts as a forwarder: every message sent from a client A to another client B is sent to the server, encrypted as

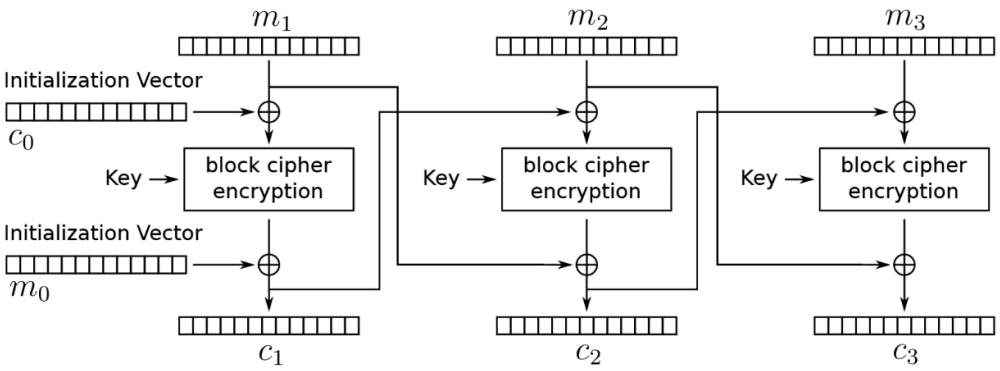


Figure 5.3: Encryption in IGE mode.

a cloud chat message shown in section 5.2 with the authorization key of A; the server then decrypts the message and encrypts it as a cloud chat message using the authorization key of B and finally sends it to B. The writing $\{M\}_{k_{AS}}$ in fig. 5.4 expresses that the message M has been encrypted with the authorization key k_{AS} using the cloud chat encryption schema section 5.2.

Let us describe the protocol steps, which are summed up in fig. 5.4. We will examine a run where Alice requests the secret chat. Notice that we assume that both Alice and Bob have already exchanged an authorization key with the server (k_{AS} and k_{BS}, respectively).

1. First of all, client A retrieves Diffie-Hellman parameters (p, g) from the server. Then, it generates a Diffie-Hellman ephemeral key a and a session identifier sID. Finally, it computes his Diffie-Hellman half key $g_a := g^a \mod p$ and sends $\langle sID, B, g_a \rangle$ to client B;

2. Client B receives the secret chat request and may either accept it or deny it. Upon accepting it, it receives p, g, generates his ephemeral key b, computes his half key $g_b := g^b \mod p$ and the newly created secret chat key $k_{AB} := g_a^b \mod p$. Lastly, it sends $\langle sID, g_b, \text{sha1}\,(k_{AB}) \rangle$ to client A. The hash function performed on the key is, as stated in the official documentation, only used as a sanity check for client developers;

3. Client A receives the half key of B g_b and uses it to compute the secret chat key $k_{AB} := g_b^a \mod p$. If the fingerprint of the key matches the one sent by B, it accepts the secret chat and can start sending and receiving messages[3].

Mind that Diffie-Hellman generator (g), ephemeral keys (a, b) and half keys (g_a, g_b) are supposed to be checked by the clients as shown in section 5.1. Additionally, they should check all these values are greater then 1 and smaller then $p-1$. Telegram team also recommends to check that $2^{2048} \leq g_a, g_b \leq (p-2)^{2064}$.

Notice that, as this exchange lacks authentication of the two clients, a compromised server can trivially perform a classic Diffie-Hellman man-in-the-middle attack [31]. To this end, Telegram requires the clients to check if their key fingerprints match. This must be done in an authenticated way using an out-of-band channel (e.g. in-person). This comparison is a necessary condition to ensure secrecy in further exchanges between participating clients.

[3]The encryption schema used to encrypt end-to-end messages is very similar to the cloud chat schema. We will not report it for brevity. The main difference is in the composition of the message. Please refer to [53] for more details.

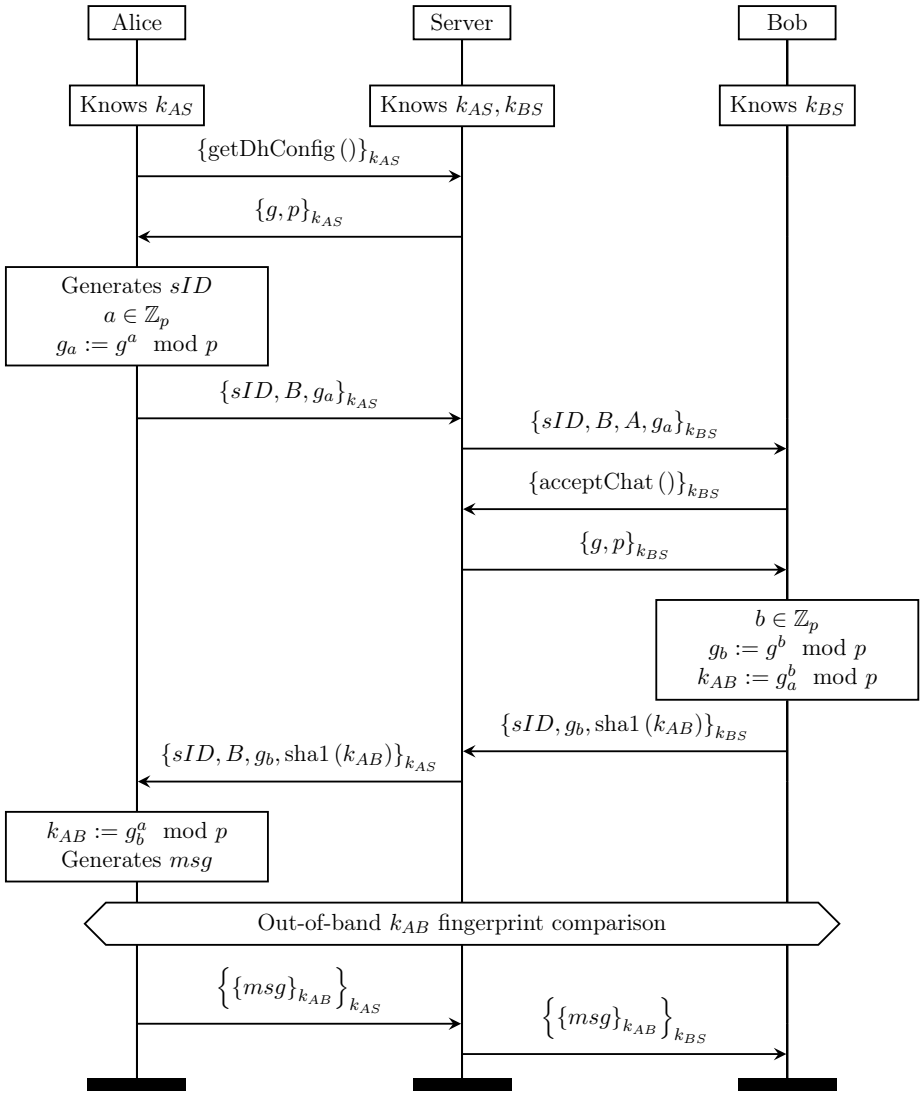


Figure 5.4: MTProto2.0 Secret chat protocol

5.4 Rekeying protocol

This protocol is used to enable perfect forward secrecy in secret chats. Clients compliant with the protocol specification are supposed to initiate rekeying once a secret chat key has been used to encrypt and decrypt more than 100 messages, or it has been in use for at least a week (and was used at least once). Any client participating in a secret chat can start a run of the rekeying protocol.

Rekeying is based on Diffie-Hellman, which ensures that access to old keys does not grant knowledge of new ones. Moreover, the exchange is encrypted with the secret chat key shared between clients, which effectively authenticates clients under the assumption of secrecy of the key itself. Under the premise of secrecy of the initial secret chat key, this also rules out the possibility of a man-in-the-middle attack executed by a compromised server.

Let us examine the rekeying protocol, represented schematically in fig. 5.5. We assume that clients engaging the rekeying protocol have already shared a secret chat key k_{AB}. In addition, clients re-use

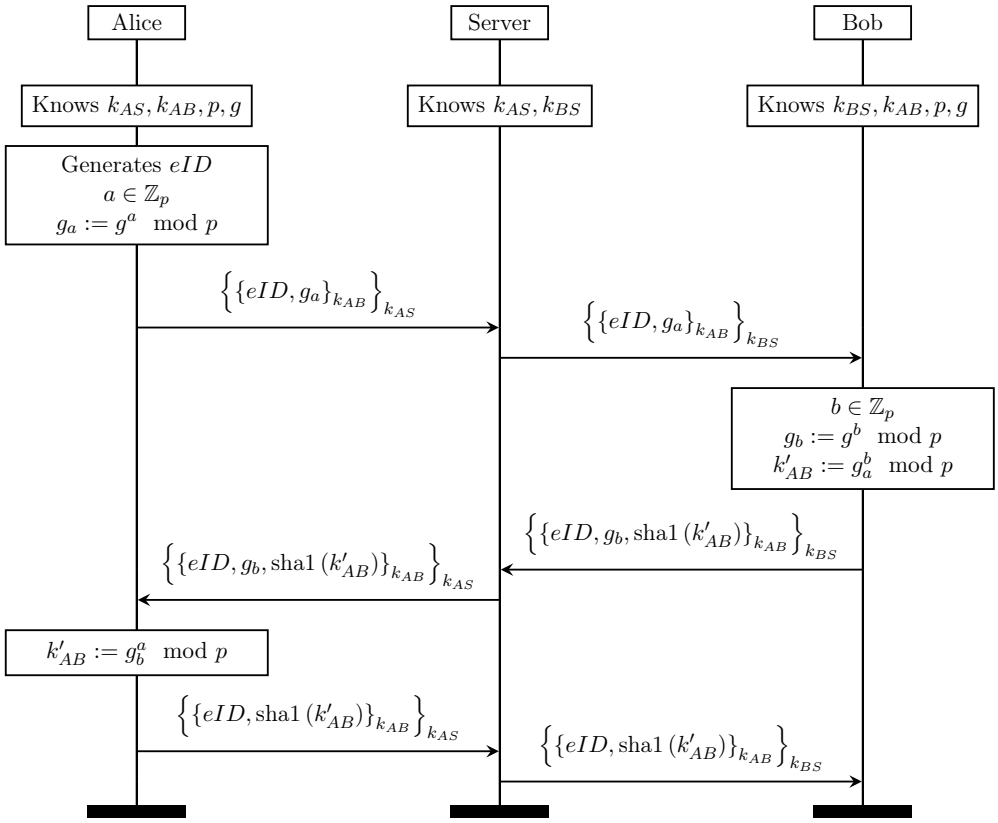


Figure 5.5: MTProto2.0 Rekeying protocol

Diffie-Hellman parameters p, g obtained from the previous secret chat protocol run.

1. Client A generates its Diffie-Hellman ephemeral key a and an exchange identifier eID. Then, it computes its half key $g_a := g^a \mod p$ and sends $\langle eID, g_a \rangle$ to client B;

2. Client B receives A's half key g_a, generates its own ephemeral key b and computes the corresponding half key $g_b := g^b \mod p$, as well as the new key $k'_{AB} := g_a^b \mod p$. Lastly, it sends $\langle eID, g_b, \text{sha1}\,(k'_{AB}) \rangle$ to client A. As it was for the secret chat protocol, the hash function performed on the new key is only used as a sanity check for client developers;

3. Client A receives the half key of B g_b and uses it to compute the new key $k'_{AB} := g_b^a \mod p$. After checking the hash, it sends the last message (basically an acknowledgement) containing $\langle eID, \text{sha1}\,(k'_{AB}) \rangle$.

6

Formalization in Tamarin prover

In this section, we will explore the implementation of the MTProto2.0 protocol created in Tamarin prover. The full code is available on GitHub [56]. Please refer to the README instructions for the code structure and for how to run the code.

This formalization is based on the paper [44] and the Proverif analysis [42] of MTProto2.0 by M. Miculan and N. Vitacolonna.

We will proceed to analyze the implementation of every single protocol and schema described in chapter 5.

6.1 Authorization protocol

6.1.1 Exchanges formalization

Let us describe how the protocol was formalized. See fig. 6.1 for the updated schematic of the protocol.

Round 1 Client nonces (both n_c and n_k) are generated as fresh terms. Diffie-Hellman parameters (p and g) are not modeled: instead, we use a single public constant `'g'` that represents the generator. In Tamarin, this is needed to avoid having lots of partial deconstructions, which were causing the non-termination of the protocol proofs. As this public constant is known to everyone (including the attacker), there is no need to send it to the client in 4^{th} message. This simplification makes even more sense if we consider that MTProto2.0 uses only six values for the generator g $(2, 3, 4, 5, 6, 7)$.

Moreover, the proof-of-work is not included as we are dealing with the protocol in a symbolic model. Finally, in our formalization, we assume that the client is able to get the public key of the server from its fingerprint. In Telegram, these keys are usually embedded in the application itself, resulting in the possibility of tampering [44]. Keypairs are generated using the following rule:

```
1  rule RegisterPublicKey:
2    let
3      pkey        = pk(~skey)
4      fingerprint = fpk(pkey)
5    in
6      [ Fr(~skey) ]
7    -->
8      [ !PrivateKey($X, ~skey), !PublicKey($X, pkey, fingerprint), Out(pkey) ]
```

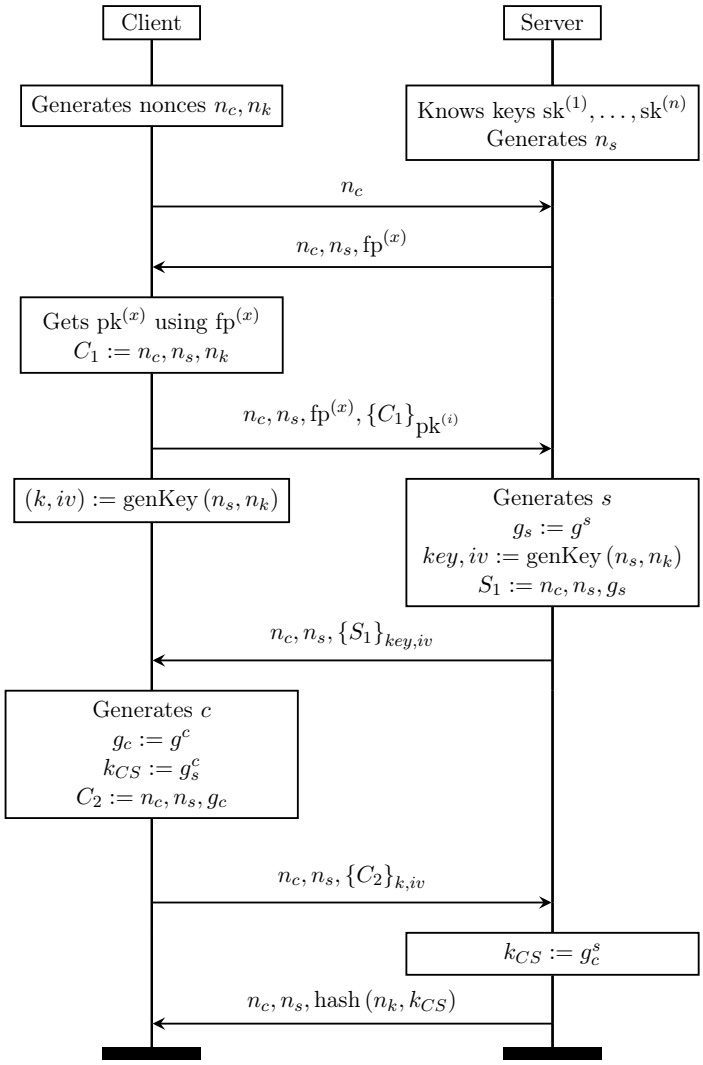


Figure 6.1: MTProto2.0 Authorization protocol (simplified)

In particular, the server decides which public key to use beforehand. Hence, a single key fingerprint is sent to the client. As the attacker cannot add its keys, this does not affect the soundness of the results.

Round 2 Another simplification is seen in the 3rd message: as encryption is not malleable in the symbolic model, there is no need to introduce the hash of the plaintext message along with the message itself. Notice that this hash was used as a Message Authentication Code (MAC) to check data integrity after decryption. This applies to every message from now on. Public key encryption is defined using the built-in `asymmetric-encryption` equational theory.

Following from the formalization in Proverif, we do not model time, and the generic key derivation function kdf() has been renamed to genKey(). Lastly, a public constant `'StoC'` (`'CtoS'`) has been used to mark the message from the server to the client (from the client to the server) that contains the server (client) Diffie-Hellman half key. This is needed to avoid adding an incorrect reflection attack in which the server receives the message he has previously sent. In reality, the encryption actually contains some data that allows matching messages. Symmetric encryption is modeled using the built-in

`symmetric-encryption` equational theory.

In the implementation, we also make substantial usage of pattern matching. Besides, using pattern matching is encouraged by Tamarin's manual as it usually decreases partial deconstructions. We have made use of it to ensure that half keys of client and server are actually in the form of `'g'^~x`. This trick improves verification times by a lot, while it leaves the man-in-the-middle attack still possible[1]. We will use this modeling trick for every Diffie-Hellman exchange, including the one used in the secret chat and rekeying protocols.

Round 3 No simplification is needed for the last round, except for removing the $retryID$, meaning that we assume that the exchange is always successful. Notice that this is consistent with the MTProto2.0 specification: the client needs to retry to send his half key only when the server finds a duplicated key hash, but, in our model, this never happens as client and server use fresh values as Diffie-Hellman ephemeral keys.

6.1.2 Additional implementation notes

Every encrypted exchange is tagged with a public constant `'AUTH_X'`, which should improve efficiency.

The server's nonce in the model might also be fixed, which models a flawed server implementation or a server lacking randomness. The following rules are used to achieve this behavior:

```
1  rule GenerateRandomServerNonce:
2      [ Fr(~ns) ]
3    -->
4      [ NS(~ns) ]
5
6  rule GenerateFixedServerNonce:
7      []
8    -->
9      [ NS('FIXED_NS') ]
```

The server then, in the multiset rewriting rule premises, uses the `NS(ns)` fact.

Finally, many compromise rules are created:

- Compromise of server long-term key (asymmetric private key)
- Compromise of client secret nonce n_k
- Compromise of server ephemeral key (DH exponent)
- Compromise of client ephemeral key (DH exponent)
- Compromise of authorization keys

As compromise rules are very simple and similar to each other, we will only show an example:

```
1  /* Reveals the client's DH secret exponent. */
2  rule CompromiseAuthProtClientExponent:
3      [ !AuthProtClientEphemeralSecrets(nk, c) ]
4    --[ CompromisedClientExponent(c) ]->
5      [ Out(c) ]
```

[1] The attacker only needs to use its own ephemeral key and send the corresponding half key.

Using these compromise rules, we can also check if the protocol is secure in the eCK model [38]. The eCK model assumes, in the case of an authenticated key exchange, two parties, each having a long-term and an ephemeral secret. Of these four pieces of information, the eCK model allows revealing any subset that does not contain both long-term and ephemeral secrets. In MTProto2.0, the authorization protocol does not respect the eCK model as the client has no long-term secret. Moreover, intuitively, we can already notice that the protocol is not secure in the eCK model: revealing the client ephemeral secret n_k allows the attacker to perform a classic man-in-the-middle attack on the Diffie-Hellman exchange [32].

6.1.3 Security properties verification

Every rule in the protocol execution is labeled with action facts. We then use these to model security properties. Following the Proverif formalization, we have modeled several forms of key agreement, authentication of parties and key secrecy, along with observational equivalence queries on the secret nonce n_k and the authorization key. In the following paragraphs, we are going to examine lemmas and related results in more detail.

Key agreement The following lemma models key agreement:

```
1  lemma LemmaAuthProtAgreement:
2    "
3      /* Whenever a client and a server negotiate an authorization key */
4      ∀ nc ns nk authKey1 authKey2 #i #j.
5        (
6          ServerAcceptsAuthKey(nc, ns, nk, authKey1) @i ∧
7          ClientAcceptsAuthKey(nc, ns, nk, authKey2) @j ∧
8
9          /* and no secret was leaked */
10         ¬(∃ sk #r.   CompromisedAuthKey(sk) @r) ∧
11         ¬(∃ skey #r. CompromisedPrivateKey(skey) @r) ∧
12         ¬(∃ n #r.    CompromisedNk(n) @r) ∧
13         ¬(∃ c #r.    CompromisedClientExponent(c) @r) ∧
14         ¬(∃ s #r.    CompromisedServerExponent(s) @r)
15       )
16       ⟹
17       (
18         /* then the authorization key is the same */
19         ( authKey1 = authKey2 ) ∨
20
21         /*
22          * or the server is actually running two different instances
23          * of the protocol with the client
24          */
25         (
26           ∃ #n1 #n2.
27             ServerGeneratesNonce(ns) @n1 ∧
28             ServerGeneratesNonce(ns) @n2 ∧
29             ¬(#n1 = #n2)
30         )
31       )
32   "
```

Lines 10-14 are used to define what attacks the attacker is not allowed to perform in this specific lemma. This technique is ubiquitous in our lemmas formalization. In Proverif, the same lemma is written without these conjunctions as the main process definition defines which attacks the adversary can carry out. By commenting any line between 10-14 (inclusive), we can model agreement in the presence of some

information leakage.

Key agreement without leaks holds. It may be interesting to notice that key agreement holds even when both ephemeral keys are revealed as the attacker cannot force the client and the server to compute different keys. Compromising one at a time either server's private key or client nonce allows the attacker to execute a man-in-the-middle attack on the Diffie-Hellman exchange.

We can also prove a similar property: if a client and a server end a run of the protocol negotiating the same key in their unrelated sessions, then these sessions actually coincide.

Authentication Client authentication in the protocol does not hold because the client does not authenticate itself and the server is willing to execute the protocol with anybody (including the attacker). This query captures this:

```
lemma LemmaAuthProtAuthClientToServer:
  "
    ∀ nc ns nk authKey #i #j.
      /* Whenever a client has started a session with nonce nc */
      ClientStartsSession(nc) @i ∧

      /* and the server has sent an ACK for the session <nc, ns> */
      ServerSendsAck(nc, ns, nk, authKey) @j ∧

      /* and no secret was leaked */
      ¬(∃ sk #r.   CompromisedAuthKey(sk) @r) ∧
      ¬(∃ skey #r. CompromisedPrivateKey(skey) @r) ∧
      ¬(∃ n #r.    CompromisedNk(n) @r) ∧
      ¬(∃ c #r.    CompromisedClientExponent(c) @r) ∧
      ¬(∃ s #r.    CompromisedServerExponent(s) @r)
      ⟹
      (
        /* then a client has shared an authKey with the server */
        (
          ∃ #k.
          ClientAcceptsAuthKey(nc, ns, nk, authKey) @k
        ) ∨

        /*
         * or the server is actually running two different instances
         * of the protocol with the client
         */
        (
          ∃ #k #l.
            ServerGeneratesNonce(ns) @k ∧
            ServerGeneratesNonce(ns) @l ∧
            ¬(#k = #l)
        )
      )
  "
```

However, we can prove the server knows that the client that negotiated the authorization key is the same that sends the third message. For the sake of brevity, we will not report the related lemma. As anyone can create an authorization key with the server, this lack of authentication is not an issue.

However, server authentication is fundamental, and it is captured by the following lemma:

```
1   lemma LemmaAuthProtAuthServerToClient:
2     "
3       ∀ nc ns nk authKey #i1.
4         /* Whenever a client receives an ACK from the server */
5         ClientReceivesAck(nc, ns, nk, authKey) @i1 ∧
6
7         /* and no secret was leaked */
8         ¬(∃ sk #r.   CompromisedAuthKey(sk) @r) ∧
9         ¬(∃ skey #r. CompromisedPrivateKey(skey) @r) ∧
10        ¬(∃ n #r.    CompromisedNk(n) @r) ∧
11        ¬(∃ c #r.    CompromisedClientExponent(c) @r) ∧
12        ¬(∃ s #r.    CompromisedServerExponent(s) @r)
13        ⟹
14        (
15          /* then there is a session matching it on the server */
16          (
17            ∃ #j.
18            ServerAcceptsAuthKey(nc, ns, nk, authKey) @j ∧
19            (∀ #i2. ClientReceivesAck(nc, ns, nk, authKey) @i2 ⟹#i1 = #i2)
20          ) ∨
21
22          /* or the server has reused the same nonce */
23          (
24            ∃ #j1 #j2.
25              ServerGeneratesNonce(ns) @j1 ∧
26              ServerGeneratesNonce(ns) @j2 ∧
27              ¬(#j1 = #j2)
28          )
29        )
30    "
```

This lemma holds, meaning that the server is authenticated to the client. Notice that line 19 models injectivity.

Key secrecy Another fundamental property of the authentication protocol is key secrecy: when a client and a server negotiate a key, they must be sure that the key is known only to them. This property is proved using the following lemma:

```
1   lemma LemmaAuthProtKeySecrecy:
2     "
3       /* Whenever client and server negotiated a key */
4       ∀ nc ns nk authKey #i #j #k.
5         ClientAcceptsAuthKey(nc, ns, nk, authKey) @i ∧
6         ServerAcceptsAuthKey(nc, nk, nk, authKey) @j ∧
7
8         /* and the attacker knows it */
9         K(authKey) @k
10        ⟹
11        /* then some secret was leaked */
12        (
13          (∃ #r.      CompromisedAuthKey(authKey) @r) ∨
14          (∃ skey #r. CompromisedPrivateKey(skey) @r) ∨
15          (∃ #r.      CompromisedNk(nk) @r) ∨
16          (∃ c #r.    CompromisedClientExponent(c) @r) ∨
17          (∃ s #r.    CompromisedServerExponent(s) @r)
18        )
19
20    "
```

Observational equivalence Attempts to prove observational equivalence for client's secret nonce n_k and authorization key, unfortunately, do not terminate. Notice that observational equivalence is often extremely resource-consuming.

Observational equivalence in Tamarin is expressed using the `diff/2` operator, which duplicates every rule, one for the left-hand side and one for the right-hand side of the operator, and tries to find a difference between the two traces.

In the formalization, we created the two following rules:

```
1  /*
2   * The secret nonce nk generated by the client is indistinguishable
3   * from a fresh value.
4   */
5  rule RuleAuthProtNkEquivalence:
6      [
7        !AuthProtClientEphemeralSecrets(nk, b),
8        Fr(~n)
9      ]
10    -->
11     [ Out(diff(nk, ~n)) ]
12
13 /*
14  * A negotiated authorization key authKey is indistinguishable from a
15  * fresh value.
16  */
17 rule LemmaAuthProtAuthKeyEquivalence:
18     [
19       !AuthKeyClient(server, authKey),
20       Fr(~freshAuthKey)
21     ]
22    -->
23     [ Out(diff(~freshAuthKey, authKey)) ]
```

6.2 Cloud chat encryption schema

6.2.1 Encryption formalization

Cloud chat encryption has been simplified to suit the symbolic model better. First of all, the key derivation function returns only the key. Notice that, as both key and initialization vector are derived from the same terms, once the adversary is able to compute the key it would be able to compute the IV as well. Hence, it is not modeled as it would only add complexity to the model without bringing any benefit.

A function `msgKey/2` is used to create the message key from the message and the authorization key. Then, the message key is used to create the encryption key for the message, together with the authorization key, using the `genKey/2` primitive. The plaintext message is encrypted using the built-in `symmetric-encryption`. The final message that is sent on the channel is composed of the fingerprint of the authorization key (using `keyID/1`), the message key and the encrypted message. Notice that functions `msgKey/2`, `genKey/2` and `keyID/1` have no associated equation (i.e. are considered perfect hashing functions).

Four different rules have been created: two are used to exchange messages from client to server and the other two for messages from the server to the client. This approach allows us to test for secrecy in

both directions. Here is an example of the rule used to model a client to server message:

```
1   rule ClientCloudChatSendsMessage:
2     let
3       msg = <'CC_CtoS', ~sessionID, ~m>
4       mk  = msgKey(msg, authKey)
5       key = genKey(mk, authKey)
6       c   = <keyID(authKey), mk, senc(msg, key)>
7     in
8       [
9         !AuthKeyClient($Server, authKey),
10        Fr(~sessionID),
11        Fr(~m)
12      ]
13    --[ ClientSendsCloudMessage(~sessionID, ~m, authKey) ]->
14      [ Out(c) ]
```

Notice on line 3 that we use a public constant `'CC_CtoS'` to differentiate client to server from server to client messages.

6.2.2 Security properties verification

Secrecy and forward secrecy The cloud chat encryption schema is essentially employed to obtain secrecy on messages exchanged between a client and a server after they have negotiated an authorization key. The following lemma proves secrecy from client to server. A similar one is used to verify secrecy of messages from server to client.

```
1  lemma LemmaCloudChatSecrecyClientToServer:
2    "
3      ∀ sid msg authKey #i #j #r.
4        (
5          /* Whenever a client sends a cloud message to the server */
6          ClientSendsCloudMessage(sid, msg, authKey) @i ∧
7
8          /* and the server receives it */
9          ServerReceivesCloudMessage(sid, msg, authKey) @j ∧
10
11         /* and the attacker knows it */
12         K(msg) @r
13       )
14       ⟹
15       (
16         /* then some secret was compromised */
17         (∃ #r.      CompromisedAuthKey(authKey) @r) ∨
18         (∃ skey #r. CompromisedPrivateKey(skey) @r) ∨
19         (∃ n #r.    CompromisedNk(n) @r) ∨
20         (
21           (∃ c #r.   CompromisedClientExponent(c) @r) ∧
22           (∃ s #r.    CompromisedServerExponent(s) @r)
23         )
24       )
25   "
```

This lemma means that messages are secure, unless:

- The private key of the server is compromised;
- The secret nonce n_k of the client is compromised;
- Diffie-Hellman exponents of both client and server are compromised.

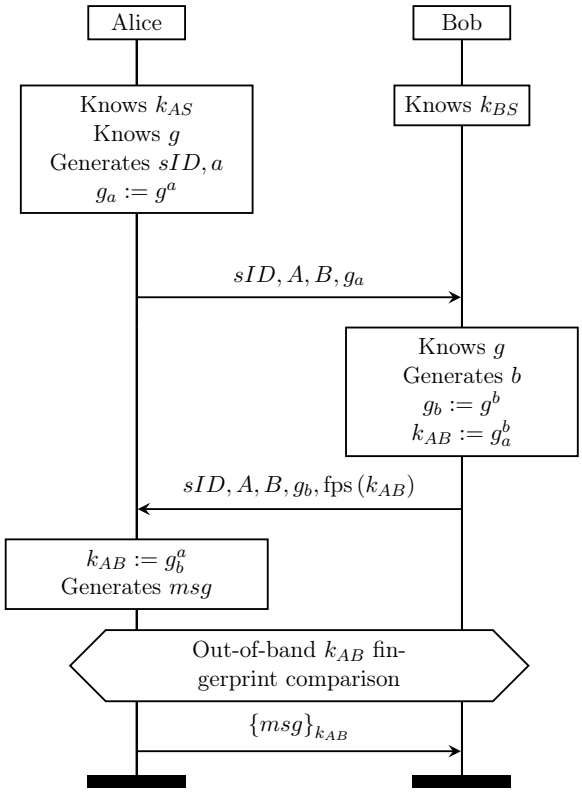


Figure 6.2: MTProto2.0 Secret chat protocol formalization

As seen in section 6.1, compromising any of these secrets leads to a lack of secrecy on the key. Additionally, the key itself can be compromised to break cloud chats secrecy.

As the lemma above is also strict (i.e. removing any action fact leads to an attack trace), this also means that perfect forward secrecy does not hold in cloud chats as an attacker that is able to compromise the authorization key can decrypt both past and future messages (as well as forging them, effectively impersonating the client to the server).

6.3 Secret chat

A secret chat can be created after clients have both shared an authorization key with the server. To create a shared secret (the *secret chat key*) between two clients, MTProto2.0 uses, as already seen in section 5.3, a Diffie-Hellman key exchange in which the Telegram server acts as a forwarder.

As we have already analyzed the security of the authorization protocol and of the cloud chat encryption in sections 6.1 and 6.2, in the formalization we do not encrypt messages with the authorization key, nor we use the server as a forwarder. Instead, we execute the secret chat protocol exchanges as if they were plaintext: this allows the attacker to act as the server (i.e. having knowledge of both authorization key and forwarding messages) and to manipulate messages [44]. Additionally, removing this layer of encryption allows simplifying the protocol model, formulating stronger security properties, and obtaining better efficiency. For these reasons, we do not model any rule that creates dummy authorization keys

nor execute the authorization protocol to obtain them.

6.3.1 Exchanges formalization

Following the Proverif formalization, we explicitly name clients and allow the attacker to choose their roles (i.e. who the initiator is, whom the initiator talks to, who the responder is). We apply restriction rules to ensure that the initiator is always honest:

```
restriction RestrictionChoosePrincipal:
  "
    ∀ X Y Z #i.
      ChoosePrincipal(X, Y, Z) @i ⟹ ((X = Y) ∨ (X = Z))
  "
```

However, the responder may as well be a dishonest party. Having named clients explicitly, we can prove additional authentication security properties.

Finally, we model the QR-code comparison with the following rules:

```
rule PerformOutOfBandKeyComparison:
    [
      !QR(aID, aUser, bUser, sessionKey),
      !QR(bID, bUser, aUser, sessionKey)
    ]
  --[
      /* Rule out the possibility of sessions of a client with itself */
      NotEq(aUser, bUser),

      OutOfBandKeyComparisonSucceeded(aID, aUser, bUser, sessionKey),
      OutOfBandKeyComparisonSucceeded(bID, bUser, aUser, sessionKey)
    ]->
    [
      !QROK(aID, aUser, bUser, sessionKey),
      !QROK(aID, bUser, aUser, sessionKey)
    ]

rule SkipOutOfBandKeyComparison:
    [ !QR(aID, aUser, bUser, sessionKey) ]
  --[ OutOfBandKeyComparisonSkipped(aID, aUser, sessionKey)]->
    [ !QROK(aID, aUser, bUser, sessionKey) ]

restriction RestrictionNotEqual:
  "∀ x y #i. NotEq(x, y) @i ⟹¬(x = y)"
```

The persistent fact `!QR` is created by each party after they have computed the secret chat key. The above rules model both the correct and incorrect behavior of clients, allowing to specify properties in which a specific behavior is taken. The correct behavior is the one express by the `PerformOutOfBandKeyComparison` rule, where the client implicitly checks that the `sessionKey` he computed is the same as the other party. The incorrect behavior in rule `SkipOutOfBandKeyComparison` models a client that does not match the key fingerprint: as this is the only way of authenticating parties to each other, skipping the comparison may result in a Diffie-Hellman man-in-the-middle attack.

Additionally, two rules have been added to allow message sending and receiving:

```
rule SecretChatSend:
  let
    mk  = msgKey(~m, sessionKey)
    key = genKey(mk, sessionKey)
    c   = <keyID(sessionKey), mk, senc(~m, key)>
  in
    [
      !SecretChatClient(X, iUser, rUser, xID, chatID, sessionKey, authKey),
      Fr(~m)
    ]
  --[
      ClientSendsSecretChatMsg(chatID, X, iUser, rUser, sessionKey, ~m)
    ]->
    [ Out(c) ]

rule SecretChatReceive:
  let
    mk  = msgKey(~m, sessionKey)
    key = genKey(mk, sessionKey)
    c   = <keyID(sessionKey), mk, senc(~m, key)>
  in
    [
      !SecretChatClient(X, iUser, rUser, xID, chatID, sessionKey, authKey),
      In(c)
    ]
  --[ ClientReceivesSecretChatMsg(chatID, X, iUser, rUser, sessionKey, ~m) ]->
    []
```

As already noted in section 5.3, the encryption schema for secret chat messages slightly differs from the cloud chat one. However, from a symbolic point of view, they are equivalent. Hence, we model the secret chat encryption with the same steps described in section 6.2.

Figure 6.2 shows the secret chat protocol, with the simplifications described above.

6.3.2 Security properties verification

Two main properties should be satisfied by the secret chat protocol: secrecy and authentication.

Secrecy This is, of course, one of the most important properties for an end-to-end chat. MTProto2.0 guarantees secrecy under the assumption that clients compare the secret chat key fingerprint out-of-band. This mechanism is also used, for example, by Signal [16]. Formally, Tamarin is able to prove the following secrecy lemma:

```
lemma LemmaSecretChatSecrecy:
  "
    /* Whenever the client sends a secret chat message msg */
    ∀ chatID X iUser rUser sessionKey msg #i #j.
      ClientSendsSecretChatMsg(chatID, X, iUser, rUser, sessionKey, msg) @i ∧

      /* but the attacker knows it */
      K(msg) @j
      ⟹
      (
        /* then clients skipped the QR validation */
        ∃ a #r. OutOfBandKeyComparisonSkipped(a, X, sessionKey) @r
      )
  "
```

In other words, unless clients skipped the key fingerprint comparison, the exchanged messages are

secret. Let us stress the fact that this result does not depend on the secrecy of the authorization key. Hence, if clients compare the fingerprint correctly, end-to-end encryption holds even against a compromised server.

Authentication Many variants of authentication have been proven. The most generic one is the following:

```
lemma LemmaSecretChatAuthentication1:
"
  /* Whenever a client received a secret chat message */
  ∀ chatID1 X iUser rUser msg sessionKey #i.
    ClientReceivesSecretChatMsg(chatID1, X, iUser, rUser, sessionKey, msg) @i
    ⟹
    (
      /* then it was sent by another (honest) client */
      (∃ Y chatID2 #r.
        ClientSendsSecretChatMsg(chatID2, Y, iUser, rUser, sessionKey, msg) @r) ∨
      (∃ Y chatID2 #r.
        ClientSendsSecretChatMsg(chatID2, Y, rUser, iUser, sessionKey, msg) @r) ∨

      /* or clients involved have skipped the QR validation */
      (
        ∃ Y xID yID #r1 #r2.
          OutOfBandKeyComparisonSkipped(xID, X, sessionKey) @r1 ∧
          OutOfBandKeyComparisonSkipped(yID, Y, sessionKey) @r2
      )
    )
"
```

Notice that we cannot match sessions (i.e. $chatID1 = chatID2$) as the server can forward messages modifying the session identifier. As also pointed out by M. Miculan and N. Vitacolonna [44], this does not seem to pose any security risks.

As reported earlier, skipping QR comparison results in a classic Diffie-Hellman man-in-the-middle attack.

Observational equivalence We have tried to model several observational equivalence properties, such as:

- Indistinguishability of a client's exponent from a random value. This should not hold as the attacker can compute the DH key (using the exponent and the other half of the key) and compare fingerprints;
- Indistinguishability of a secret chat key and a random value. This should not hold as the attacker can compare the key fingerprint with the 2^{nd} message of the protocol;
- Indistinguishability of message exchanged with the secret chat key and a random value. This should hold as long as the key used to encrypt the message is kept secret.

Again, Tamarin does not terminate when trying to prove these observational equivalences.

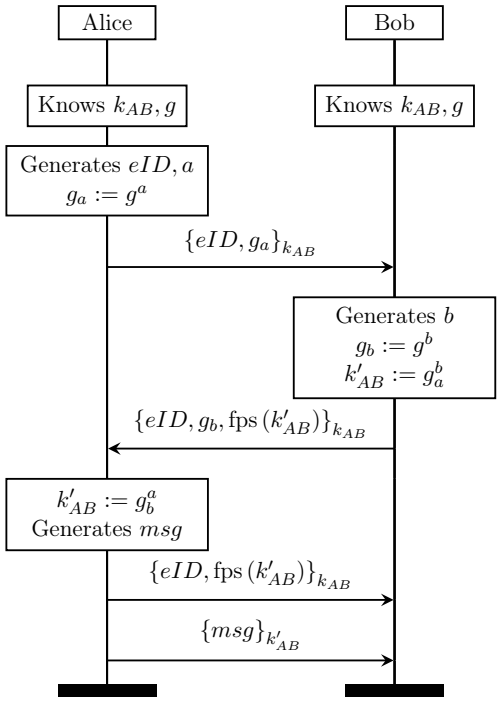


Figure 6.3: MTProto2.0 Rekeying protocol formalization

6.4 Rekeying

Finally, let us describe the rekeying protocol formalization, which allows perfect forward secrecy on secret chats.

6.4.1 Exchanges formalization

The formalization of the rekeying protocol is very similar to the formalization of the secret chat protocol: we do not model the outer layer of encryption from client to server. The reasoning and implications are the same (see section 6.3).

The significant difference is that this time the Diffie-Hellman exchange is end-to-end encrypted using the secret chat key exchanged (only) between clients. A schematic representation of the protocol's formalization is given in fig. 6.3.

For efficiency and termination reasons, in our formalization the attacker can choose only the initiator's responder. Both initiator and responder are fixed. Secret chat keys are generated using the following rules:

```
rule GenerateSecretChatKey:
  let
    X1 = 'Alice'
    X2 = 'Bob'
    sessionKey = 'g' ^ ~i ^ ~r
  in
    [
      Fr(~i),
      Fr(~r),
      Fr(~chatID)
    ]
  -->
    [
      !SecretChatClient(X1, X1, X2, ~i, ~chatID, sessionKey),
      !SecretChatClient(X2, X1, X2, ~r, ~chatID, sessionKey)
    ]

rule AttackerGeneratesSecretChatKey:
  let
    sessionKeyAlice = 'g' ^ ~e1 ^ ~e2
    sessionKeyBob = 'g' ^ ~e3 ^ ~e4
  in
    [
      Fr(~e1), Fr(~e2), Fr(~e3), Fr(~e4),
      Fr(~chatIDAlice), Fr(~chatIDBob)
    ]
  --[ AttackerRegisteredKey(sessionKeyAlice), AttackerRegisteredKey(sessionKeyBob) ]->
    [
      !SecretChatClient('Alice', 'Alice', 'Eve', ~e1, ~chatIDAlice, sessionKeyAlice),
      !SecretChatClient('Eve', 'Alice', 'Eve', ~e2, ~chatIDAlice, sessionKeyAlice),
      !SecretChatClient('Bob', 'Eve', 'Bob', ~e3, ~chatIDBob, sessionKeyBob),
      !SecretChatClient('Eve', 'Eve', 'Bob', ~e4, ~chatIDBob, sessionKeyBob),
      Out(<sessionKeyAlice, sessionKeyBob>)
    ]
```

The first models the generation for honest users, while the second allows the attacker to execute the protocol with honest users by creating a secret chat with them and sharing the key with the attacker by outputting it. Notice that these rules implicitly assume the correctness of the secret chat protocol, which we have proven in section 6.3.

A crucial modeling decision has been taken to achieve termination of lemmas. To do so, we have devised the following method: both initiator's and responder's first rule have a restriction `OnlyTwice(x)`, where `x` is `'Initiator'` for initiator and `'Responder'` for responder. The restriction is defined as follows:

```
restriction RestrictionOnlyTwice:
  "
    ∀ x #i #j #k.
      OnlyTwice(x) @i ∧
      OnlyTwice(x) @j ∧
      OnlyTwice(x) @k
      ⟹
      (
        #i = #j ∨
        #i = #k ∨
        #j = #k
      )
  "
```

This restriction allows only two instances of the `OnlyTwice` fact (with the same argument) on the trace. Nonetheless, this is the best solution that has been found to the non-termination problems observed

during the modeling of this protocol[2]. It is of the utmost importance to notice that this restriction may or may not compromise the soundness of results obtained in the verification process as it bounds the number of rekeying executions.

Message encryption is modeled as in secret chats (see section 6.3).

6.4.2 Security properties verification

We would like to prove three properties: secrecy, perfect forward secrecy and authentication.

Secrecy and perfect forward secrecy Even in our scenario, it can be proven by Tamarin that secrecy holds after rekeying. The following lemma formalizes this property:

```
lemma LemmaRekeyingMessageSecrecy:
  "
    /* There is no way two honest clients exchanged a message msg */
    ¬(∃ s r iUser rUser xID1 xID2 eID1 eID2 newKey msg #i1 #i2 #j.
      ClientSendsMessageWithNewKey(xID1, s, eID1, iUser, rUser, newKey, msg) @i1 ∧
      ClientReceivesMessageWithNewKey(xID2, r, eID2, iUser, rUser, newKey, msg) @i2 ∧

      /* and the attacker knows it */
      K(msg) @j
    )
  "
```

Perfect forward secrecy is guaranteed by the periodic rotation of keys described in section 5.4. We also proved that knowledge of authorization keys does not compromise secrecy.

Authentication The other fundamental property of this protocol is the authentication of parties. Whenever two clients execute the rekeying protocol (successfully), they should be able to assume that the new key is known to both and only them.

The following lemma is used to verify that whenever two clients exchange the same key in the same session, there are only two (honest) parties involved:

```
  lemma LemmaRekeyingAuthentication:
  "
    /* Whenever two users negotiate the same key in the same session */
    ∀ exchangeID iUser1 iUser2 rUser1 rUser2 newKey #i #j.
      InitiatorHasNegotiatedNewKey(exchangeID, iUser1, rUser1, newKey) @i ∧
      ResponderHasNegotiatedNewKey(exchangeID, iUser2, rUser2, newKey) @j
      ⟹
      (
        /* then there are actually only two users involved */
        iUser1 = iUser2 ∨
        iUser1 = rUser2 ∨
        rUser1 = iUser2 ∨
        rUser1 = rUser2
      )
  "
```

However, Tamarin finds a counterexample for this lemma. Specifically, it finds the same attack described by M. Miculan and N. Vitacolonna: an *unknown key-share* attack.

[2]In essence, Tamarin was trying to gain knowledge of secret (i.e. exponents) or encrypted terms by executing the rekeying protocol many times with different secret chat keys. This behavior appears to lead to non-termination.

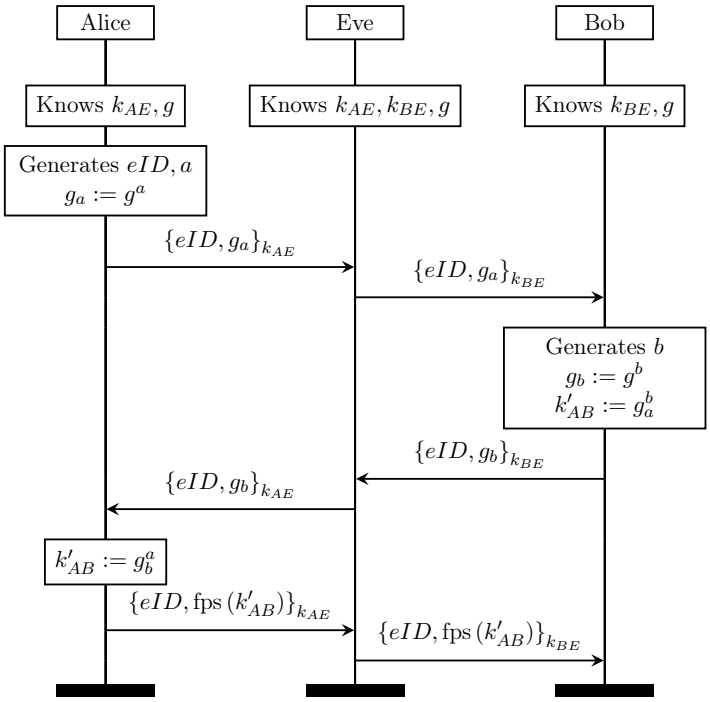


Figure 6.4: Unknown key-share attack on the rekeying protocol.

Let A and E be two honest entities. This type of attack, defined by S. Blake-Wilson and A. Menezes [10], allows a client A to believe it has shared a key with E, while it has instead shared it with B. In this particular case, this attack also compromises two other properties: *implicit key confirmation* and *implicit key authentication* [44]. The first property states that client A is assured that the second entity B can compute the key. The latter states that client A is assured that no other entity, apart from B, can learn the value of the key [10].

Now, let us show how the attack unfolds in the following situation: assume there are three clients (A, B and E) and client E has already shared a secret chat key. The steps of the attack are the following:

1. A starts executing the rekeying protocol with E and E receives A's half key g_a;
2. E now starts a concurrent execution with B using A's half key g_a and obtains B's half key g_b. E may or may not reuse the same session identifier sent by A with B;
3. E responds to A using B's half key g_b;
4. A concludes the protocol run by sending the fingerprint of the new key. E then forwards, with the correct key, the fingerprint.

As a result, client E neither uniquely possesses the key nor can compute it. Figure 6.4 shows a graphical representation of the attack.

7

Comparison

In this chapter, we will compare Tamarin prover and Proverif under several points of view. In particular, the comparison is going to cover the following topics:

- Usability
- Expressiveness
- Efficiency
- Soundness and completeness

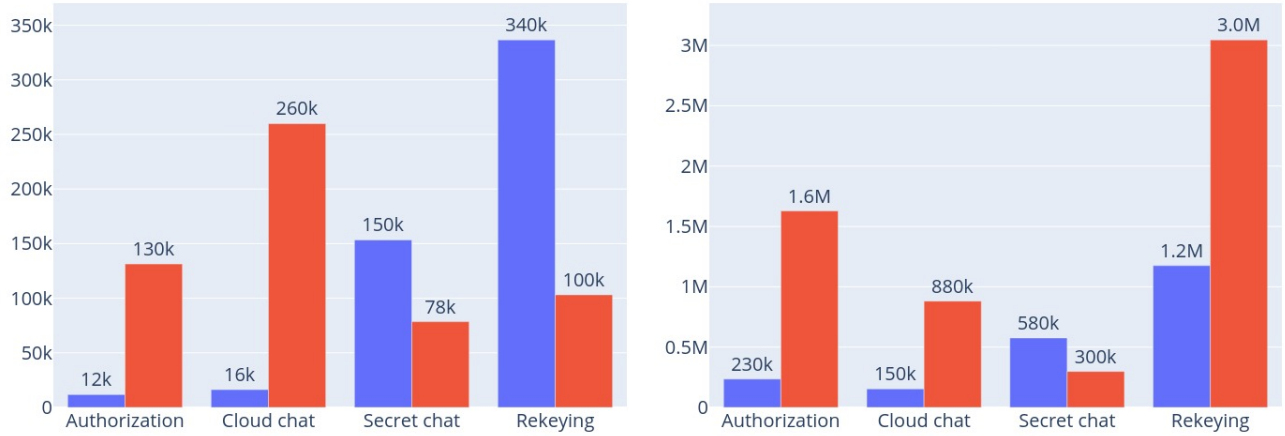
7.1 Usability

This may be the most complex property to evaluate as perceived usability can change with different levels of expertise and previous backgrounds. Of course, a user who has already used the π-calculus and Prolog would probably feel more comfortable with Proverif's formalisms. The usability of these tools for the formal verification of cryptographic protocols is, in general, difficult to establish. To the best of our knowledge, no study has been published regarding this specific topic. Another issue regards the formalization in Proverif, which was created by M. Miculan and N. Vitacolonna, meaning we cannot know the problems that arose during modeling.

As we have seen, Tamarin offers a wide variety of built-in cryptographic primitives. While these are not hard to define, having predefined primitives allows for faster modeling compared to Proverif (where every single primitive must be defined).

The labeled multiset rewriting rules used by Tamarin is also more "low-level" than the applied π-calculus. As such, modeling using the latter is usually easier. A variant of the applied π-calculus, Sapic, has also been created to allow protocol specification in a stateful version of the applied π-calculus which is then automatically translated to labeled multiset rewriting rules and analyzed using Tamarin [36]. The authors' intentions were to create a sound and complete verification tool that was not "error-prone and difficult" as Tamarin, while keeping its features intact [35].

The security properties of both languages should be straightforward to model as these are based on well-known logical constructs. However, the guarded fragment of first-order logic used by Tamarin has a more complex syntax, which can probably intimidate beginners at first.



(a) Wall clock time (ms)

(b) Maximum resident set size (kB)

Figure 7.1: In blue Tamarin, in red Proverif.
Wall clock time and maximum resident set size for every protocol and schema.

As hinted in section 2.1, Tamarin prover also offers the possibility of using interactive proofs. The tool orders the facts to resolve with the chosen heuristics and allows the user to choose which to solve first. This sometimes leads to more efficient proofs in the case of `exists-trace` lemmas. Tamarin also offers the possibility of using an oracle (i.e. a script receiving and sending data to standard input and output) to automatize the interactive proof. Please refer to the user manual for further information.

Assuming no previous knowledge, we can conclude that Tamarin's formalism and language might be slightly more complex than Proverif's.

7.2 Expressiveness

The various built-in theories offered by Tamarin are also very expressive. As we have seen in section 3.2.4, the Diffie-Hellman, the xor, and the bilinear-pairing equational theories are modeled faithfully. Moreover, Tamarin is one of the few tools able to model exclusive or [24].

The labeled multiset rewriting rules used by Tamarin is a more low-level formalism than then applied π-calculus. Nonetheless, rewriting rules allow encoding stateful protocols, which are not supported by the Horn clauses used by Proverif. Moreover, as reported by S. Kremer and K. Robert [36], many other efficient tools, such as AVISPA [7] or Maude-NPA [27], fail to analyze protocols that require a non-monotonic global state (that is, some type of memory that can be read and altered). Proverif only supports the `table` construct to save some information, but this construct does not support removal.

7.3 Efficiency

In this section, we will compare the efficiency of Tamarin's and Proverif's MTProto2.0 formalization. Of course, the measured efficiency is indicative of this implementation only: as a small change can lead to both great performances or non-termination, this is only to be intended as the obtained efficiency of the current version of both implementations.

We consider two different metrics: *maximum resident set size* (in kB) and *wall clock time* (in ms).

For the methodology, we followed the same approach of P. Lafourcade and M. Puys [37]. Benchmarks were run on an Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz CPU with 16 GB of RAM. To get these three metrics we used the `time` command (GNU version) [33]. This version of time uses the `getrusage` Linux system call, which returns resource usage measures for a process, including our metrics. For each tool and protocol, we executed the proof 40 times. The entire process takes about 12 hours, depending on the hardware. For benchmarks, we used Proverif 2.02pl1 and Tamarin prover 1.6.0. Plots in fig. 7.1 show the mean value of each execution, while table 7.1 shows many relevant statistics. As we can see, the maximum resident set size is usually significantly lower for Tamarin, while the wall clock time differs from protocol to protocol.

	Peak memory size (kb)		**Time (ms)**		
	Tamarin	Proverif	Tamarin	Proverif	
Mean	234707.60	1628285.70	11726.00	131329.50	**Auth**
Deviation	4965.62	149.06	381.48	198.86	
Median	234696	1628286	11740	131270	
Min	226000	1627968	10880	131140	
Max	245500	1628620	12330	131990	
Mean	153411.70	881417.40	16290.50	259920.50	**Cloud chat**
Deviation	4133.22	123.44	238.54	469.46	
Median	152588	881388	16280	260015	
Min	146384	881244	15760	259040	
Max	163840	881628	16960	261000	
Mean	576006.80	298242.10	153397.75	78498.50	**Secret chat**
Deviation	20908.28	118.18	674.64	170.40	
Median	576940	298234	153465	78500	
Min	529172	298032	151200	78220	
Max	618464	298488	154770	78940	
Mean	1175407.70	3046324.80	336571.25	103154.25	**Rekeying**
Deviation	14630.59	206.79	820.91	73.00	
Median	1175464	3046334	336415	103140	
Min	1138496	3045988	334970	103050	
Max	1212192	3046780	338460	103440	

Table 7.1: Efficiency of the two models for every protocol.

7.4 Soundness and completeness

Let us define what soundness and completeness of formal verification tools mean. Definitions are the same given by M. Furer et al. [30]. *Soundness* means that any statement that can be proven is valid. A proof system is sound if whenever we can prove something, it is also true. As a formula:

$$\Gamma \vdash Q \quad \Longrightarrow \quad \Gamma \models Q \tag{7.1}$$

Completeness means that the proof system is powerful enough to prove any valid statement (in some class). In other words, a proof system is complete if whenever something is true, it can be proved to be so. As a formula:

$$\Gamma \models Q \quad \Longrightarrow \quad \Gamma \vdash Q \tag{7.2}$$

As we can see, soundness is a critical property. Unsound proof systems allow proofs of false statements. In the case of cryptographic protocol verification, this may imply, in the best-case scenario, that the prover yields a proof for a false attack. In the worst case, the tool may miss an attack, which is far more problematic. Completeness, on the contrary, is a nice-to-have property. Incompleteness means there are some true properties for which there are no proofs in our formal system. [5].

The foundational papers of Proverif and Tamarin prover respectively assert that:

- Proverif is sound, but not complete [14];
- Tamarin is both sound and complete [50, 51, 24].

It is important to note that both soundness and completeness are relative to the properties that a tool is able to model in its language. For example, Proverif might be incomplete simply because it can cover a larger family of properties. To the best of our knowledge, no formal expressivity comparison of Tamarin and ProVerif has been done so far, so we do not know whether the set of properties which we can describe in Tamarin is larger, included, or unrelated to those of Proverif.

8

Conclusions

We have shown the two models for cryptographic protocol verification and we have also described the symbolic model. We have shown how Tamarin prover and Proverif can be used to model a simple protocol in chapter 4. Then, we have presented the MTProto2.0 protocol suite and its formalization using Tamarin prover. We have proved its security in our fully automatic model, and we have re-discovered that the protocol is vulnerable to a theoretical unknown key-share (UKS) attack. This attack allows a malicious client B, with the help of another client E, to convince client A that she has shared a key with E, while, instead, she has shared it with B. Finally, we have compared both Tamarin prover and Proverif implementations and found that they have comparable efficiency. However, Tamarin prover excels on the expressiveness, especially considering the powerful built-in theories (such as Diffie-Hellman) available to the user.

Overall, we were able to reproduce the same results obtained with Proverif using Tamarin prover, with some simplifications and restrictions.

Bibliography

[1] Martín Abadi, Bruno Blanchet, and Cédric Fournet. Just fast keying in the pi calculus. *ACM Trans. Inf. Syst. Secur.*, 10(3):9–es, July 2007.

[2] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, page 104–115, New York, NY, USA, 2001. Association for Computing Machinery.

[3] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. *SIGPLAN Not.*, 36(3):104–115, January 2001.

[4] Martín Abadi and Bruno Blanchet. Computer-assisted verification of a protocol for certified email. *Science of Computer Programming*, 58(1):3–27, 2005. Special Issue on the Static Analysis Symposium 2003.

[5] Butterfield Andrew. Formal methods. `https://www.scss.tcd.ie/Andrew.Butterfield/Teaching/CS3001/`. Accessed: 2021–09-23.

[6] Hajnal Andréka, István Németi, and Johan van Benthem. Modal languages and bounded fragments of predicate logic. *Journal of Philosophical Logic*, 27(3):217–274, 1998.

[7] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P. C. Heám, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The avispa tool for the automated validation of internet security protocols and applications. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification*, pages 281–285, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[8] Alessandro Armando, Roberto Carbone, and Luca Compagna. Satmc: A sat-based model checker for security-critical systems. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 31–45, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[9] Gregory Bard. Modes of encryption secure against blockwise-adaptive chosen-plaintext attack. *IACR Cryptology ePrint Archive*, 2006:271, 01 2006.

[10] Simon Blake-Wilson and Alfred Menezes. Unknown key-share attacks on the station-to-station (sts) protocol. In *Public Key Cryptography*, pages 154–170, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[11] Blanchet Bruno. Using Horn Clauses for Analyzing Security Protocols. In Véronique Cortier and Steve Kremer, editors, *Formal Models and Techniques for Analyzing Security Protocols*, volume 5 of *Cryptology and Information Security Series*, pages 86 – 111. IOS Press, 2011.

[12] Blanchet Bruno. Security protocol verification: Symbolic and computational models. In Pierpaolo Degano and Joshua D. Guttman, editors, *Principles of Security and Trust*, pages 3–29, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[13] Blanchet Bruno. 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.

[14] Blanchet Bruno, Smyth Ben, Cheval Vincent, and Sylvestre Marc. *Automatic Cryptographic Protocol Verifier, User Manual and Tutorial*, 2020.

[15] Blanchet Bruno and Cheval Vincent. Proverif: Cryptographic protocol verifier in the formal model. `https://prosecco.gforge.inria.fr/personal/bblanche/proverif/`.

[16] 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, 2017.

[17] Véronique Cortier, Steve Kremer, and Bogdan Warinschi. A Survey of Symbolic Methods in Computational Analysis of Cryptographic Systems. Research Report RR-6912, INRIA, 2009.

[18] Cas J. F. Cremers. The scyther tool: Verification, falsification, and analysis of security protocols. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, pages 414–418, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[19] Basin David, Cremers Cas, Dreier Jannik, and Sasse Ralf. Security protocol analysis using the tamarin prover. Accessed: 2021–08-27, 2017.

[20] Basin David, Cremers Cas, Dreier Jannik, Meier Simon, Sasse Ralf, and Schmidt Benedikt. Tamarin prover. `http://tamarin-prover.github.io/`.

[21] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

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

[23] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *The Journal of Logic Programming*, 1(3):267–284, 1984.

[24] Jannik Dreier, Lucca Hirschi, Sasa Radomirovic, and Ralf Sasse. Automated unbounded verification of stateful cryptographic protocols with exclusive or. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 359–373, 2018.

[25] Nancy Durgin, Patrick Lincoln, and John Mitchell. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12:247–311, 02 2004.

[26] Rescorla Eric. The transport layer security (tls) protocol version 1.3. https://tlswg.org/tls13-spec/draft-ietf-tls-rfc8446bis.html. Accessed: 2021-08-20.

[27] Santiago Escobar, Catherine Meadows, and José Meseguer. *Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties*, pages 1–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[28] S. Even and O. Goldreich. On the security of multi-party ping-pong protocols. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 34–39, 1983.

[29] Cedric Fournet. Hiding names: Private authentication in the applied pi calculus. 04 2003.

[30] Martin Furer, Oded Goldreich, and Yishay Mansour. On completeness and soundness in interactive proof systems. 1989.

[31] Aaron C. Geary. Analysis of a man-in-the-middle attack on the diffie-hellman key exchange protocol. https://calhoun.nps.edu/handle/10945/4509, 2009.

[32] Aaron C. Geary. Analysis of a man-in-the-middle attack on the diffie-hellman key exchange protocol, 2009.

[33] GNU. time(1) - linux man page. https://linux.die.net/man/1/time.

[34] Alfred Horn. On sentences which are true of direct unions of algebras. *Journal of Symbolic Logic*, 16(1):14–21, 1951.

[35] Inria. Sapic - a stateful applied pi calculus. http://sapic.gforge.inria.fr/.

[36] Steve Kremer and Künnemann Robert. Automated analysis of security protocols with global state. Research report, March 2014.

[37] Pascal Lafourcade and Maxime Puys. Performance Evaluations of Cryptographic Protocols Verification Tools Dealing with Algebraic Properties. In *8th International Symposium on Foundations and Practice of Security 8th International Symposium, FPS 2015*, pages 137–155, Clermont-Ferrand, France, October 2015. Springer.

[38] Brian LaMacchia, Kristin Lauter, and Anton Mityagin. Stronger security of authenticated key exchange. In Willy Susilo, Joseph K. Liu, and Yi Mu, editors, *Provable Security*, pages 1–16, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[39] Chris M. Lonvick and Tatu Ylonen. The Secure Shell (SSH) Protocol Architecture. RFC 4251, January 2006.

[40] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[41] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[42] Miculan Marino and Vitacolonna Nicola. Formal verification of telegram chat protocol mtproto 2.0. https://github.com/miculan/telegram-mtproto2-verification/tree/alt-formalization, 2020.

[43] Simon Meier. Advancing automated security protocol verification. 2013.

[44] Marino Miculan and Nicola Vitacolonna. Automated symbolic verification of telegram's mtproto 2.0. *CoRR*, abs/2012.03141, 2020.

[45] Robin Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, USA, 1999.

[46] James A. Muir. Seifert's rsa fault attack: Simplified analysis and generalizations. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *Information and Communications Security*, pages 420–434, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[47] Kobeissi Nadim, Bhargavan Karthikeyan, and Blanchet Bruno. Automated verification for secure mes- saging protocols and their implementations: A symbolic and computational approach. *2nd IEEE European Symposium on Security and Privacy*, pages 435–450, 2017.

[48] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, December 1978.

[49] Michaël Rusinowitch and Mathieu Turuani. Protocol insecurity with finite number of sessions is np-complete. In *Proceedings of the 14th IEEE Workshop on Computer Security Foundations*, CSFW '01, page 174, USA, 2001. IEEE Computer Society.

[50] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. Automated analysis of diffie-hellman protocols and advanced security properties. In *2012 IEEE 25th Computer Security Foundations Symposium*, pages 78–94, 2012.

[51] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. Automated analysis of diffie-hellman protocols and advanced security properties. In *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium*, CSF '12, page 78–94, USA, 2012. IEEE Computer Society.

[52] The Tamarin Team. *Tamarin-Prover Manual: Security Protocol Analysis in the Symbolic Model*, 2021.

[53] Telegram. End-to-end encryption, secret chats. https://core.telegram.org/api/end-to-end. Accessed: 2021–08-30.

[54] Telegram. Faq for the technically inclined. https://core.telegram.org/techfaq#q-do-you-use-ige-ige-is-broken. Accessed: 2021–08-30.

[55] Telegram. Mtproto mobile protocol. https://core.telegram.org/mtproto/.

[56] Alessandro Zanatta. Mtproto2.0-tamarin. https://github.com/AlessandroZanatta/MTProto2-Tamarin, 2021.