

Automated Symbolic Verification of the Needham-Schroeder Protocol

Denis D'Ambrosi¹

Abstract

As cryptographic protocols become both more complicated and more critical by the day, we must find a way to systematically verify message exchanges without the added burden of manual proving. To solve this issue, multiple theorem provers have been proposed in the last years, each one with its flaws and advantages. In this paper, we deal with the Tamarin prover, an automated tool that exploits multiset rewriting rules and guarded fragments of temporal logic to model cryptographic protocols and check their security properties. After an initial introduction to the topic as a whole, we dig deeper into the tool from a user's perspective and show how a simple key exchange protocol (the symmetric Needham-Schroeder protocol) can be modeled within a logic theory. This example will allow us to show how easily some message exchanges can be formalized within this prover; on the other hand, it will also demonstrate how the unbounded nature of this tools will require us to take some further assumptions on the protocol in order to avoid non-termination.

Keywords

Network Security — Automated Verification — Formal Verification

¹Student number 147681, dambrosi.denis@spes.uniud.it

Contents

Introduction	1
1 The symbolic model	1
1.1 Limits of the symbolic model	2
2 Tamarin Prover Overview	2
2.1 Basic definitions	2
2.2 Term algebra	3
2.3 Default facts and rules	4
2.4 Trace properties	5
2.5 Observational equivalence properties	5
2.6 Aiding termination	6
Source lemmas • Oracles • Interactive mode • Restrictions • Re-use lemmas	
3 The Needham-Schroeder Protocol	6
3.1 Tamarin Formalization	7
The protocol • Termination and properties • Performance metrics	
4 Conclusions	14
References	14

Introduction

Security protocols are three-line programs that people still manage to get wrong

Roger M. Needham

As computing becomes more ubiquitous and distributed,

we need an ever-increasing amount of cryptographic protocols to allow different parties to share information in a confidential, authenticated and integral way. New communication protocols are being deployed by the day, but most of them have at least some hidden flaw that, once found, can compromise the security of the communication, with potentially catastrophic effects. To avoid such a disaster, our best defense is to actually verify said protocols in some formal model, that will ensure us that, at least within the assumptions of the model itself, the exchange can not be exploited maliciously. Writing these proof by hand is often a cumbersome, long and error-prone process, thus in the last years computer scientists and mathematicians have instead started relying on automated tools that can carry on the demonstration for them. In this essay, we will take a closer look at one of such tools (the Tamarin prover [Tea13]), by presenting its syntax, strengths, limitations and a practical example of modelization.

The essay is structured as follows: in section 1 we will briefly introduce one of such formal verification models, in section 2 we will give an introduction to the Tamarin prover from a user's perspective, while in the following section we will show how such tool can be used for real-world formal verification by modeling the Needham-Schroeder Symmetric protocol. Finally, in section 4 we will summarize the results of this paper.

1. The symbolic model

Known also as the *Dolev Yao* model, the symbolic model abstracts from cryptography by substituting real-world cryp-

tographic operations with term-algebras [DY83]. Encryption schemes are formalized through binary isomorphisms within $\{0, 1\}^*$: given a symmetrical key k and the relevant pair of cryptographic encryption and decryption primitives senc_k and sdec_k , symmetrical cryptography is defined through the following identity:

$$\text{sdec}_k \text{senc}_k = 1 \quad (1)$$

Only the entities that are in possess of k are able to encrypt or decrypt any message with it [Bru12]: this assumption is known as perfect cryptography. Furthermore, given a message M and its image $\text{senc}_k M$, we assume that it is impossible for an attacker who does not know k to:

- guess or bruteforce k ;
- manipulate $\text{senc}_k M$
- infer any information about M from $\text{senc}_k M$.

Similarly to equation 1, asymmetric encryption and signing are modeled through the following identity:

$$\text{adec}_{pr} \text{aenc}_{pub} = \text{adec}_{pub} \text{aenc}_{pr} = 1 \quad (2)$$

In this case, we do not consider a single key used for both encryption and decryption k , but instead an entangled pair of keys pr and pub . The assumptions for cryptography are the same as above.

A protocol is modeled as a series of formal terms (messages) exchanged by abstract machines (clients) through an attacker-controlled network. We assume that any connection can be eavesdropped by a malevolent agent that is also able to intercept, modify, forge and drop messages on the fly, with the only exception of the encryption constraints previously described. After formalizing a protocol, its security goals can be expressed as [BBB⁺19]:

- Trace properties: invariants that should hold for each possible execution of the protocol;
- Observational equivalence properties: properties that an attacker should not be able to distinguish between two different runs of the protocol.

The use of algebraic terms within the Dolev Yao model allows for simple abstractions during formal checking. As opposed to the aforementioned standard model, where bitstrings take the place of terms and perfect cryptography is not an assumption, the symbolic model is less realistic since it is able to guarantee only high-level security goals on the conjecture of flawless implementations of the cryptographic primitives actually used.

1.1 Limits of the symbolic model

In order to streamline the proof of security properties belonging to a certain protocol, multiple automatic tools were proposed [BBB⁺19]. These softwares, in order to prove security goals, have to compute the set of terms that an attacker is able to deduce while the protocol is executing, although since this set can possibly be infinite, the general problem suffers of both undecidability and infinite state space. To convince ourselves that this is the case, we just need to think about the fact that we can have an infinite number of sessions for each protocol, each one with different nonces and messages of unlimited size. Without bounding at least two of these sources of infinity, termination can not be guaranteed by any tool in this category.

In particular, as shown by N. Durgin et al. [DLM04], by restricting different sources we can ensure to reduce insecurity to different complexity classes:

- if we limit the number of nonces and messages, we end up with a DEXP-complete problem;
- if we restrict the number of sessions (and thus the number of nonces and messages), the problem will still be affected from infinite state space, but its solution will be NP-complete; this workaround was adopted by a variety of tools such as Cl-AtSe [Tur06], OFMC [BMV05] and SATMC [AC04].

Alternatively to bounding the sources of infinity, some software address undecidability by requiring human input (for example, Tamarin Prover [MSCB13] features an interactive mode that allows the user to decide which security goals prioritize based on intermediate constraint systems), returning inconclusive results (for example, ProverIf [Bla16] may return an invalid attack trace) or even allowing non-termination.

2. Tamarin Prover Overview

The automatic tool we used to complete the experimentation for this thesis is Tamarin Prover (which from now on we will call Tamarin for the sake of simplicity). We will now do a quick overview of its syntax, semantics and internal functioning; please refer to S. Meier's [Mei13] and B. Schmidt's [Sch12] PhD thesis and their introductory paper [SMCB12] for further information about the theoretical foundations regarding this tool, as long as its official manual [Tea22] to resolve any doubts about its use.

2.1 Basic definitions

Tamarin allows protocol (and adversary) modeling through multiset rewriting rules.

Definition 2.1: Multiset rewriting rule

Given a multiset $\Gamma_t = \{F_0, \dots, F_n\}$ and a sequence of multisets $\text{trace}_t = \langle a_0, \dots, a_{t-1} \rangle$ at a time t , we can define a rewrite rule as a triple of multisets $RR = \langle L, A, R \rangle$

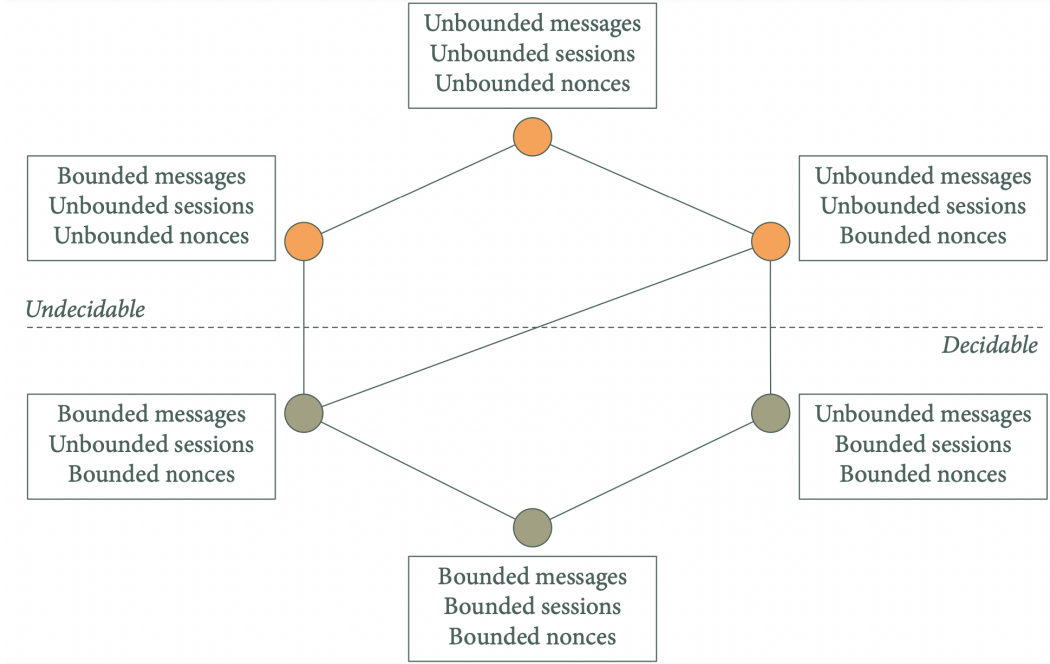


Figure 1. Decidability in symbolic verification. Image taken by N. Vitacolonna's presentation [Vit21]

(written as $RR = L - [A] \rightarrow R$) such that:

- we can apply RR to Γ_t if there is at least one ground instance (i.e. an instance with no variables) $rr = l - [a] \rightarrow r$ of RR so that $l \subseteq^\# \Gamma_t$
- applying rr to Γ_t yields to a new state Γ_{t+1} and an increased trace $trace_{t+1}$ obtained as

$$\Gamma_{t+1} = \Gamma_t \setminus^\# l \cup^\# r \quad (3)$$

$$trace_{t+1} = \langle a_0, \dots, a_{t-1}, a_t \rangle \quad (4)$$

in which $\setminus^\#$ and $\cup^\#$ are the multiset equivalent operations for set difference and union. From now on, we will refer to L , R and A as the multisets of *premises*, *conclusions* and *action facts* of a rule (in this order).

Note that within the trace $trace_{t+1}$, a is associated to the time instant t : this will be useful later on since security properties will be specified through first order logic guarded formulas. To be precise, a *trace* is the sequence of sets of ground (action) facts that happened during the execution of a series of rewriting rules.

More formally, given a set of rules P , all the possible traces generated by executions of P are

$$traces(P) = \{ \langle A_1, \dots, A_n \rangle \mid \exists S_1, \dots, S_n. \emptyset^\# \xrightarrow{A_1} S_1 \xrightarrow{A_2} \dots \xrightarrow{A_n} S_n \text{ and no ground instance of Fresh() is used twice} \}$$

For the purpose of this thesis is also worth mentioning that since in a rewriting rule $L - [A] \rightarrow R$ any of the three

multisets could be empty, a trace in which all instances of $\emptyset^\#$ are removed is called an *observable trace*.

Within Tamarin, all multisets are composed by facts. Facts feature a precise arity, are named with a starting capital letter and can be either *linear* or *persistent*:

- linear facts can be consumed only once and are useful to model state transitions and ephemeral messages;
- persistent facts can be consumed unlimited times and are optimal to model enduring knowledge (and are prefixed by an exclamation mark). Note that the introduction of persistent facts requires a slight change to equation 3: whereas before all elements of l were removed from Γ_t , now only the linear facts belonging to l will be subtracted, while persistent facts will never be consumed.

In turn, facts are made up of terms, which can be public (prefixed by a $\$$), freshly created (prefixed by a \sim) or normal messages (note that public and fresh terms are sub-categories of normal messages).

2.2 Term algebra

In order to continue explaining Tamarin's functioning, we have to take a look at the theoretical foundations behind the algebraic terms introduced in 1.

Definition 2.2: Signature

A signature Σ is a finite set of functions of defined arity.

Definition 2.3: Σ -terms

iven a signature Σ and a set of variables χ , with $\Sigma \cap \chi = \emptyset$ we can define the set of Σ -terms $\mathcal{T}_\Sigma(\chi)$ as the minimal set such as

- $\chi \subseteq \mathcal{T}_\Sigma(\chi)$
- $t_1, \dots, t_n \in \mathcal{T}_\Sigma(\chi) \wedge f/n \in \Sigma \implies f(t_1, \dots, t_n) \in \mathcal{T}_\Sigma(\chi)$

Note that f/n indicates a function f of arity n .

Definition 2.4: Substitution

iven a signature Σ and a set of variables χ , with $\Sigma \cap \chi = \emptyset$, a substitution is a function $\sigma : \chi \rightarrow \mathcal{T}_\Sigma(\chi)$.

Similarly, given a function $f/n \in \Sigma$, a substitution can be extended to a mapping $\sigma' : \mathcal{T}_\Sigma(\chi) \rightarrow \mathcal{T}_\Sigma(\chi)$ such that

$$f(t_1, \dots, t_n)\sigma' = f(t_1\sigma, \dots, t_n\sigma)$$

Definition 2.5: Equation over Σ

iven a signature Σ , a set of variables χ , with $\Sigma \cap \chi = \emptyset$, an equation over Σ is a pair of terms (t, u) with $t, u \in \mathcal{T}_\Sigma(\chi)$. Note that in this case, the equation would be written $t \simeq u$. In order to break loops while simplifying terms, equations can be oriented (as $t \rightarrow u$).

By introducing a set of equations E we can create a congruence relation $=_E$ on terms t and, consequently, equivalence classes $[t]_E$. The congruence relationship is known as an *equational theory*. Introducing equational theories allows us to unify terms based on the quotient algebra $\mathcal{T}_\Sigma(\chi)/_{=E}$: two terms $t, u \in \chi \cup \mathcal{T}_\Sigma(\chi)$ are equal in modulo E if and only if they belong to the same class:

$$t =_E u \iff [t]_E = [u]_E$$

Definition 2.6: (Σ, E) -Unification

iven a signature Σ , a set of variables χ , with $\Sigma \cap \chi = \emptyset$ and an equational theory E , two terms $t, u \in \mathcal{T}_\Sigma(\chi)$ are (Σ, E) -unifiable if there is at least a mapping σ such that $t\sigma =_E u\sigma$.

Normally, unification modulo theories is undecidable [SS89]: thus Tamarin recommends users to only define *subterm convergent theories* to ensure termination, even though the software never actually checks this constraint.

Definition 2.7: Convergent Theory

efore defining what a convergent theory is, we have to define both *terminating* and *confluent* theories:

- A terminating theory is an equational theory which ensures that each term has a *normal form* that can be reached through an arbitrary (but finite) number of substitutions.
- A confluent theory is an equational theory that ensures that if a term t can be rewritten as both terms t_1 and t_2 , then there must be a fourth term t' that can be reached through an arbitrary number of substitutions from both t_1 and t_2 .

A convergent theory is an equational theory that is both terminating and convergent.

Definition 2.8: Subterm Convergent Theory

subterm convergent theory is an equational theory that is convergent and, for each equation $e : L \rightarrow R, e \in E, R$ is either ground and in normal form or a proper subterm of L .

Tamarin uses equational theories to model algebraically the cryptographic primitives, such as symmetrical/asymmetrical encryption/decryption, hashing, signing, pair construction/deconstruction, elevation to power, and more. 10 built-in theories are provided to allow the user to easily formalize most cryptographic protocols:

1. hashing
2. asymmetric-encryption
3. signing
4. revealing-signing
5. symmetric-encryption
6. diffie-hellman
7. bilinear-pairing
8. xor
9. multiset
10. reliable-channel

2.3 Default facts and rules

Tamarin provides 4 built-in special facts that allow to model the generation and transmission of messages through the network:

- $\text{Fr}(\sim \text{msg})$ allows to create new messages msg . Note that each value is freshly generated (let us recall that Tamarin allows an infinite number of nonces), so the same ground value can not appear in two different instances of $\text{Fr}(\sim \text{msg})$.
- $\text{K}(\text{msg})$ indicates that the attacker deduced msg from the network.
- $\text{Out}(\text{msg})$ allows machines to send messages to the network.
- $\text{In}(\text{msg})$ allows machines to query the network for incoming messages.

These facts are defined through default multiset rewriting rules according to Dolev Yao's assumptions:

- $[\] \text{ -- } [\] \text{ --> } [\ \text{Fr}(\sim \text{msg}) \]$ allows for the generation of new fresh values.
- $[\ \text{Fr}(\sim \text{msg}) \] \text{ -- } [\] \text{ --> } [\ \text{K}(\sim \text{msg}) \]$ allows for the generation of new fresh values by the attacker.
- $[\ \text{Out}(\text{msg}) \] \text{ -- } [\] \text{ --> } [\ \text{K}(\text{msg}) \]$ allows the attacker to eavesdrop all messages travelling through the network.
- $[\ \text{K}(\text{msg}) \] \text{ -- } [\ \text{K}(\text{msg}) \] \text{ --> } [\ \text{In}(\text{msg}) \]$ allows user to retrieve messages from the attacker-controlled network.
- $[\] \text{ -- } [\] \text{ --> } [\ \text{K}(\$x) \]$ allows the attacker to discover all public names.
- $[\ \text{K}(x_1, \dots, x_n) \] \text{ -- } [\] \text{ --> } [\ \text{K}(f(x_1, \dots, x_n)) \]$ allows the attacker to apply n -ary functions to arguments he already knows.

2.4 Trace properties

As previously anticipated, in Tamarin security properties are expressed as guarded fragments of first order logic. The syntactical construct adopted by Tamarin to indicate a property to prove is `lemma`. Lemmas can be either of type `exists-trace`, which are demonstrated by finding a single valid protocol trace verifying the formula, or `all-traces`, which are proved by negating the property and trying to find a counterexample. To specify a lemma, we can combine the following atoms:

- `false` \perp ;
- logical operators $\neg \wedge \vee \implies$;
- quantifiers and variables $\forall \exists a b c$;
- term equality $t_1 \approx t_2$;
- time point ordering and equality $i < j$ and $i = j$;
- action facts at time points $F @ i$ (for an action fact F at timepoint i).

Keeping in consideration that also properties can define sets of traces similarly to protocol rules, we can define correctness as follows:

Definition 2.9: Correctness

iven a set of protocol rules P and a property to prove ϕ , P is correct in regards to ϕ if and only if the set of traces generated by P is a subset of the one generated by ϕ :

$$P \models \phi \iff \text{traces}(\phi) \subseteq \text{traces}(P)$$

On the contrary, if $P \not\models \phi$, all traces belonging to $\text{traces}(P) \setminus \text{traces}(\phi)$ represent valid attacks.

Note that property proving is done by Tamarin through *constraint systems* resolving. In turn, these systems are simplified by executing a backwards search within the relative protocol's *dependencies graph* (a graph that, for each fact used by a Tamarin *theory*, describes the list of possible rules usable to obtain it). The theoretical foundations behind property proving are beyond the scope of this thesis, but are explained in detail in [Sch12] [Mei13]. As end users, we can be satisfied by knowing that Tamarin's verification algorithm, although does not guarantee termination, is both *sound* and *complete* [Sch12] [Mei13] [DHRS18].

2.5 Observational equivalence properties

When operated in *Observational Equivalence Mode* (refer to the official documentation [Tea22] for further reference), Tamarin allows for the use of the `diff/2` operator [BDS15], that permits to demonstrate *diff equivalence* properties. To understand what a *diff equivalence* property is, we have to firstly introduce the concept of *trace equivalence*

Definition 2.10: Trace equivalence

wo different protocols P_1, P_2 are trace equivalent if and only if for each trace of P_1 exists a trace of P_2 so that the messages exchanged during the two executions are indistinguishable.

This is the weakest (and thus most suitable for security verification) form of observational equivalence. Since many problems in this category are undecidable in nature [CD09], Tamarin only allows *diff* equivalences in aid of termination.

Definition 2.11: Diff equivalence

wo protocols P_1, P_2 are *diff-equivalent* if and only if they have the same structure and differ only by the messages exchanged.

Thus, the two protocol have the same structure dur-

ing execution.

The previously introduced `diff(left, right)` operator allows to define a protocol rule that can be instantiated either with the `left` or `right` value. For each one of them, Tamarin constructs the relative dependencies graph and checks if they are equivalent, which is a sufficient criterion for observational diff-equivalence. Please note that a formal definition of *dependencies graph equivalence*, along with the subsequent demonstration of sufficiency are available in the introductory paper for the observational equivalences in Tamarin written by D. Basin, J. Dreier and R. Sasse [BDS15].

2.6 Aiding termination

Since protocol verification is an inherently undecidable problem, Tamarin provides some additional functionalities engineered to aid termination.

2.6.1 Source lemmas

As explained before, Tamarin elaborates the dependencies graph related to a theory before beginning constraint solving. Since the prover uses an untyped system, in certain cases it is not able to deduce the source of one or more facts, causing partial deconstructions. As explained by V. Cortier [CDD20], for example, this situation occurs whenever the same message has to travel across the network multiple times. To mitigate this issue, Tamarin allows to define a lemma with the additional tag `[sources]`, which is automatically proved during the precomputation phase and allows to declare the origin of one or more messages.

An example of this type of lemma, along with its explanation, will be provided in section 3.1.2

Note that V. Cortier, S. Dealune and J. Dreier proposed an algorithm for automatic source lemma generation [CDD20] that has already been integrated in Tamarin (run the program with the additional `--auto-sources` tag to execute the extension), but sometimes it leads to non-termination during the precomputation phase.

2.6.2 Oracles

During constraint-solving, Tamarin uses its built-in *smart* heuristic (refer to the relevant section of the manual [Tea22] for further reference) to sort the list of security goals to check. Sometimes, though, the algorithm prioritizes the wrong goals, leading to loops in the search and thus to non-termination. In an attempt to prevent this behavior, Tamarin provides also two variants of the *consecutive* heuristic, which guarantee to avoid delaying any goal infinitely to exclude starvation. This causes bigger proofs and still fails to solve the problem sometimes.

By knowing how to manually guide the search (for example after doing some practice with the built-in interactive mode), we can code (in any programming language of choice) an external *oracle*, which will be automatically run by the prover to determine the right priority objective to solve within a list of current goals. This user-defined software receives the name of the lemma and the indexed goal list (sorted by

the smart heuristic) as input, and returns the re-ranked list (or, alternatively, only its first element) as output. Note that oracles are generally stateless: for each step of the proof, the program is executed from scratch, with only the name of the considered lemma and the list of current security goals as input.

2.6.3 Interactive mode

The Tamarin interactive mode is a powerful feature that enables users to guide the proof search process and interactively inspect the demonstration as it is being constructed. When using the interactive mode, users can provide hints and guidance to the Tamarin prover, which can significantly speed up the proof search process and help to identify any potential issues or vulnerabilities in the security protocol being analyzed. For example, by interleaving automatic steps and manual selection of goals to solve, the user can understand if the tool is entering an infinite loop of computation or not by monitoring whether recursive structures of terms are being produced or not. Furthermore, by guiding the proof, the consumer can easily guess how to build an effective oracle to speed up lengthy proofs. Additionally, the interactive mode provides a valuable opportunity to gain a deeper understanding of how the Tamarin prover works and how it constructs proofs for security protocols. An example of Tamarin in action in interactive mode is displayed in figure ??.

2.6.4 Restrictions

Similarly to lemmas, *restrictions* are specified through guarded fragments of first order logic. The difference between the two is that while lemmas express properties to prove, a restriction limits the possible traces of a protocol to the executions that verify the specified formula.

2.6.5 Re-use lemmas

Finally, the last functionality we introduce are *re-use lemmas*: defined with the `[reuse]` keyword, these formulas, once proved, can be used by Tamarin in the demonstration of the subsequent specified lemmas.

3. The Needham-Schroeder Protocol

The Needham-Schroeder protocol is a key exchange protocol that was introduced in 1978 by Roger Needham and Michael Schroeder [NS78] to address the issue of authentication in distributed computer networks using symmetric key cryptography.

The Needham-Schroeder protocol is used to establish a shared symmetric key between two parties in a networked environment. The exchange assumes that both parties already have a shared secret with a trusted third party, referred to as a key distribution center (KDC). The KDC is responsible for generating and distributing secret keys to each party, and it also has a public key that can be used to verify the authenticity of messages sent by the KDC.

In the Needham-Schroeder protocol, when one party (Alice) wants to communicate securely with another party (Bob),

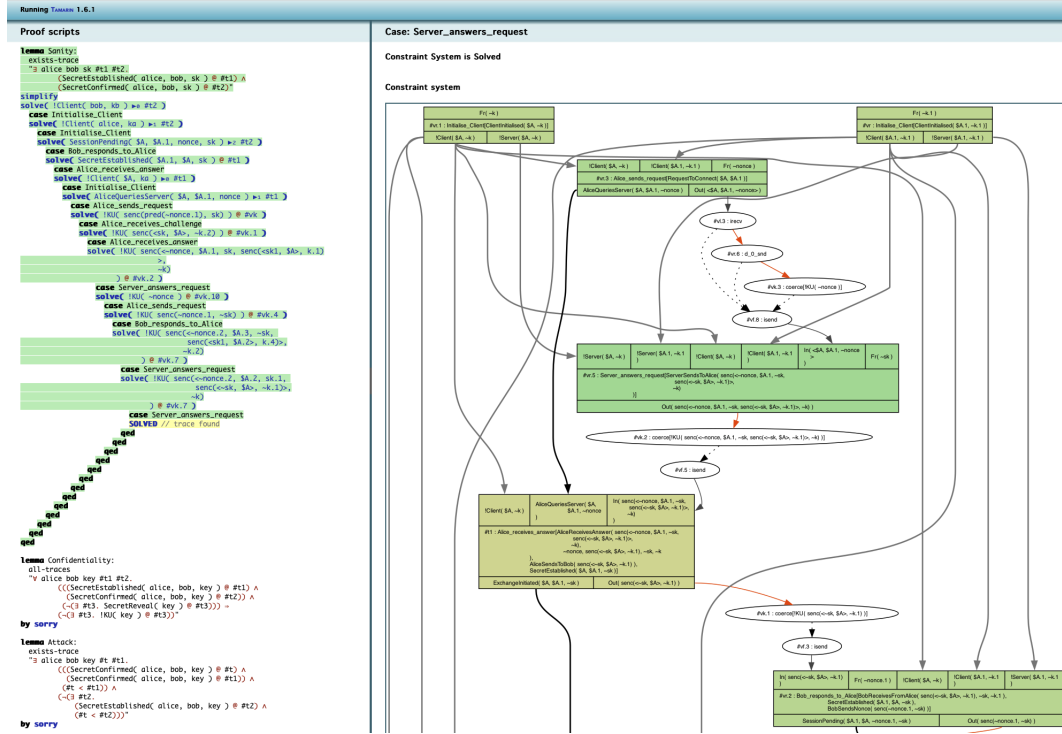


Figure 2. Tamarin's interactive mode

she sends a request to the KDC for a session key to use in the communication. The KDC then generates a new session key and sends it to Alice encrypted using Bob's public key. Alice can then decrypt the session key using her shared secret with the KDC, and use it to communicate with Bob.

The protocol also includes a mechanism for mutual authentication, which ensures that each party is who they claim to be. When Alice receives the session key from the KDC, she sends a message to Bob that includes the session key and a nonce (a random number). Bob then encrypts the nonce with the session key and sends it back to Alice. Alice can then decrypt the message and verify that the nonce matches the one she sent to Bob. This process ensures that both parties have successfully authenticated each other, and they can begin communicating securely using the session key. The exchange is summarized in diagram ??.

While the Needham-Schroeder protocol is widely used in computer security, it is not without its flaws. One issue is the potential for a "replay attack," where an attacker intercepts and reuses a previously transmitted message to gain access to the network. Another issue is the reliance on a trusted third party, which can be a single point of failure if compromised.

To address these issues, Needham and Schroeder later extended their protocol to include public-key cryptography, resulting in the Needham-Schroeder Public-Key Protocol. This protocol addressed the weaknesses of the original protocol by using public-key cryptography to establish the authenticity of each party's public key and ensure the confidentiality and integrity of the communication.

3.1 Tamarin Formalization

3.1.1 The protocol

Formalizing the Needham-Schroeder exchange in Tamarin is quite straightforward thanks to the built-in functions and simple syntax provided by the tool (note that all the source files of this section are available for local verification at [D'A23]).

Initialisation The clients (Alice and Bob) and the server are initialised with the shared secrets through the following rule:

```
1 rule Initialise_Client :
2   [
3     Fr (~k)
4   ]
5   --[ ClientInitialised($A, ~k) ]->
6   [
7     !Client($A, ~k),
8     !Server($A, ~k$)
9   ]
```

Note that since we do not have any particular reason to believe that identities are private, we must formalize them as public names known to the attacker. To avoid that the same client gets instantiated twice with two different keys, we exploit the `ClientInitialised` action-fact in the following restriction:

```
1 restriction EachClientCanBeInitialisedOnce :
2   "All client key1 key2 #t1 #t2 .
3     ClientInitialised(client, key1) @ #t1 &
4     ClientInitialised(client, key2) @ #t2
5     ==> #t1 = #t2"
```

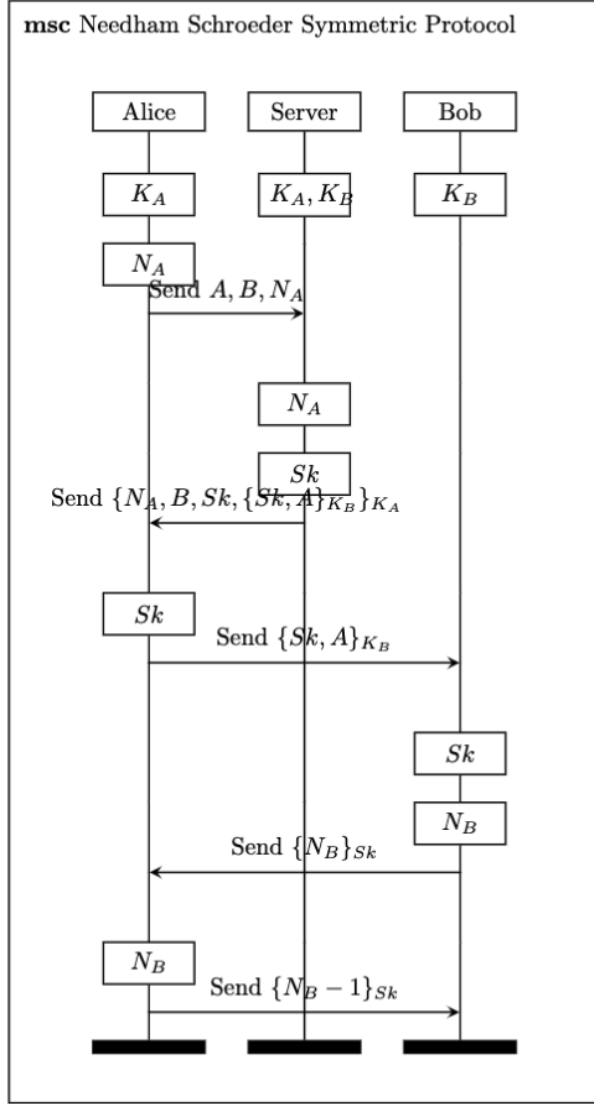


Figure 3. Needham-Schroeder Symmetric protocol

$A \rightarrow S : A, B, N_A$ The first message sent by Alice to initiate the protocol is easily modelled through the following rule (and restriction):

```

1 rule Alice_sends_request :
2   [
3     !Client(alice, ka),
4     !Client(bob, kb),
5     Fr(~nonce)
6   ]
7   --[ RequestToConnect(alice, bob) ]->
8   [
9     AliceQueriesServer(alice, bob, ~nonce),
10    Out(<alice, bob, ~nonce>)
11  ]
12
13 restriction NoSelfCommunication:
14   "All client #t .
15   RequestToConnect(client, client) @ #t
16   <- ==> F"

```

There are multiple elements that we should analyze within this snippet:

- The restriction `NoSelfCommunication` exploits the `RequestToConnect` action fact to restrict connections to only different parties. This is done to prevent Tamarin to waste resources in analyzing self-communicating runs of the protocol, which would be useless in a key-exchange scenario.
- This rule also models the generation of the nonce N_A .
- The fact `AliceQueriesServer` will then be useful, once Alice will receive the server's response, to ensure that she was actually the one which initiated the protocol with a fresh nonce before.

$S \rightarrow A : \{N_A, B, Sk, \{Sk, A\}_{K_B}\}_{K_A}$ Similarly, the role of the server in the exchange can be modelled through a single rule:

```

1 rule Server_answers_request :
2   let
3     message_to_bob = senc(<~sk, alice>, kb)
4     message_to_alice = senc(<nonce, bob, ~sk
5   <- , message_to_bob>, ka)
6   in
7   [
8     !Server(alice, ka),
9     !Server(bob, kb),
10    !Client(alice, ka),
11    !Client(bob, kb),
12    In(<alice, bob, nonce>),
13    Fr(~sk)
14  ]
15  --[ ServerSendsToAlice(message_to_alice) ]->
16  [
17    Out(message_to_alice)
18  ]

```

Here we exploit the symbolic function primitive `senc` included in the built-in `symmetric-encryption` equational theory to model the encrypting procedure executed by the server. The action-fact `ServerSendsToAlice` will be later used within a source lemma to solve Tamarin's partial deconstruction problem within this theory.

$A \rightarrow B : \{Sk, A\}_{K_B}$ Here we formalize the message sent from Alice to Bob containing the shared secret key generated by the server.

```

1 rule Alice_receives_answer :
2   let
3     message_to_bob = senc(<sk1, alice>, k)
4     message_to_alice = senc(<nonce, bob, sk,
5   <- message_to_bob>, ka)
6   in
7   [
8     !Client(alice, ka),
9     AliceQueriesServer(alice, bob, nonce),
10    In(message_to_alice)
11  ]
12  --[ AliceReceivesAnswer(message_to_alice,
13  <- nonce, message_to_bob, sk, ka),
14  AliceSendsToBob(message_to_bob),

```



```

13     SecretEstablished(alice, bob, sk) ]->
14   [
15     ExchangeInitiated(alice, bob, sk),
16     Out(message_to_bob)
17   ]

```

Similarly to before, the `AliceQueriesServer` fact is consumed to produce an analogous `ExchangeInitiated` fact, that will be later on used to keep a copy of the shared key and to ensure that Alice actually initiated an exchange if the subsequent messages will come through.

Note that, due to Tamarin’s ability to manage an unbounded number of messages, here we had to take a little shortcut in the formalization to force termination during proofs: Alice is not actually able to check the content of the `message_to_bob`, since she does not have a copy of his private key K_B , but here we use pattern matching to ensure that at least her identity is part of a message structured according the protocol specification. This assumption does not allow us to prove the security properties of the protocol without a little loss of generality, but it is crucial to avoid that Tamarin tries to find an attack trace with an infinite nested sequences of messages. Note that we did not enforce any pattern matching on K_B and Sk to avoid taking any further assumption.

Again, the `AliceReceivesAnswer` and `—AliceSendsToBob` action-facts will be exploited in the final sources lemma, while `SecretEstablished` will be useful to specify the security properties later on.

$B \rightarrow A : \{N_B\}_{Sk}$ Now we can model the challenge of Bob to Alice:

```

1 rule Bob_responds_to_Alice :
2   let
3     message_from_alice = senc(<sk, alice>,
4     ↪ kb)
5     encrypted_nonce = senc(~nonce, sk)
6   in
7   [
8     In(message_from_alice),
9     Fr(~nonce),
10    !Client(alice, ka),
11    !Client(bob, kb),
12    !Server(bob, kb)
13  ]
14  --[ BobReceivesFromAlice(message_from_alice,
15  ↪ sk, kb),
16    SecretEstablished(bob, alice, sk),
17    BobSendsNonce(encrypted_nonce) ]->
18  [
19    SessionPending(bob, alice, ~nonce, sk),
20    Out(encrypted_nonce)
21  ]

```

Following the structure of the previous rules, we can see that the formalization of this part of the exchange is pretty straightforward. The `SessionPending` fact is useful to keep track of the run of the protocol from the point of view of Bob, while the `BobReceivesFromAlice` and `BobSendsNonce` action fact are necessary to ensure solving partial deconstructions. Finally, this `SecretEstablished` fact will be used to specify a “sanity check” of the protocol via an `exists-trace` lemma.

$A \rightarrow B : \{N_B - 1\}$ We still need to formalize the last exchange. In order to model the answer to Bob’s challenge, we need to define a custom function to have the predecessor function at our disposal within the theory: the following snippet will provide both such definition (that, due to Tamarin’s syntax will be kept at the top of the file) and the subsequent rule for formalizing this step of the protocol.

```

1 function: pred/1
2
3 rule Alice_receives_challenge :
4   let
5     nonce_received = senc(nonce, sk)
6     nonce_sent = senc(pred(nonce), sk)
7   in
8   [
9     !Client(alice, ka),
10    ExchangeInitiated(alice, bob, sk),
11    !Client(bob, kb),
12    In(nonce_received)
13  ]
14  --[ AliceReceivesNonce(nonce_received, nonce
15  ↪ , sk) ,
16    AliceSendsNonce(nonce_sent) ]->
17  [
18    !Session(alice, bob, sk),
19    Out(nonce_sent)
20  ]

```

Again, if we modeled both the predecessor and the successor function, the theory would become undecidable and the security properties would not be unprovable since Tamarin would get stuck while trying to build infinite sequences of predecessors and successors.

Note that within the premises of this rule we consume the fact `ExchangeInitiated` to replace it with a persistent `!Session` between Alice and Bob. The action facts contained in the previous snippet will allow us to specify (yet again) the source lemma.

Conclusion The last protocol rule enable us to formalize the establishment of the session from Bob’s perspective:

```

1 rule Bob_receives_challenge_answer :
2   let
3     received_nonce = senc(pred(nonce), sk)
4   in
5   [
6     !Client(bob, kb),
7     !Client(alice, ka),
8     SessionPending(bob, alice, nonce, sk),
9     In(received_nonce)
10  ]
11  --[ BobReceivesNonce(received_nonce, nonce,
12  ↪ sk),
13    SecretConfirmed(alice, bob, sk) ]->
14  [
15    !Session(bob, alice, sk)
16  ]

```

Note the use of pattern matching to enforce the fact that Alice (or the attacker) must send the predecessor of the nonce previously communicated in order to trigger this transition. The `SecretConfirmed` action-fact will be required to spec-

ify the end of the execution in the following lemmas, while `BobReceivesNonce` will be part of the source lemma.

Modelling the attacker Soon we will provide an example of a possible attack to the protocol, but to do so we first need to give to the attacker the capability of deducing a previously used key from the network:

```
1 rule Secret_reveal :
2   [ !Session(client1, client2, sk) ]
3   --[ SecretReveal(sk) ]->
4   [ Out(sk) ]
```

3.1.2 Termination and properties

Enforcing termination In the previous paragraphs we introduced multiple action-facts aimed at aiding termination: such elements will now be part of a single, global source lemma:

```
1 lemma types [sources] :
2   "(All #t sk kb message .
3     BobReceivesFromAlice(message, sk, kb) @
4     #t ==>
5     (Ex #t1 . AliceSendsToBob(message) @ #t1
6     #> )
7     | (Ex #t1 . KU(sk) @ #t1 & KU(kb) @ #t1)
8     #> )
9     &
10    (All #t sk ka message nonce message_bob .
11      AliceReceivesAnswer(message, nonce,
12        #> message_bob, sk, ka) @ #t ==>
13        (Ex #t1 . ServerSendsToAlice(message) @
14        #t1)
15        | (Ex #t1 . KU(sk) @ #t1 & KU(ka) @ #t1
16          & KU(nonce) @ #t1 & KU(message_bob)
17          @ #t1))
18        &
19        (All #t sk ennonce nonce .
20          AliceReceivesNonce(ennonce, nonce, sk)
21          #> @ #t ==>
22          (Ex #t1 . BobSendsNonce(ennonce) @ #t1)
23          | (Ex #t1 . KU(sk) @ #t1 & KU(nonce) @ #t1)
24          #> t1)
25          &
26          (All #t sk ennonce nonce .
27            BobReceivesNonce(ennonce, nonce, sk) @
28            #t ==>
29            (Ex #t1 . AliceSendsNonce(ennonce) @ #t1)
30            #> t1)
31            | (Ex #t1 . KU(sk) @ #t1 & KU(nonce) @ #t1)
32            #> t1)
33            | (Ex #t1 . KU(ennonce) @ #t1))"
```

Such lemma is automatically proved by Tamarin during the precomputation phase and its statement is later on used in the proofs of the other properties to reduce drastically the number of possible sources for some facts (and, thus, sometimes allowing or even speeding-up termination).

Unfortunately, we need to add another restriction to the theory to enforce termination:

```
1 restriction NoSendingPrivateKeys :
2   "(All client key #t .
3     ClientInitialised(client, key) @ #t
4     ==> not (Ex #t1 . KU(key) @ #t1))"
```

This snippet is crucial since it allows Tamarin to get rid of all the possible attack traces in which it tries to deduce the private keys of the various parties. This restriction does not cause a loss of generality since the keys are never actually sent on the network and instead are always used only for encryption, thus the assumptions of the symbolic model will guarantee us that there is no way for the attacker of deducing such secrets.

Properties The first thing we need to check is whether our formalization of the protocol allows for a successful run of the exchange. This can be verified through an `exists-trace` lemma:

```
1 lemma Sanity :
2   exists-trace
3   "(Ex alice bob sk #t1 #t2 .
4     SecretEstablished(alice, bob, sk) @ #t1
5     #> &
6     SecretConfirmed(alice, bob, sk) @ #t2)"
```

Tamarin quickly auto-verifies this snippet and provides a valid trace modeling a correct run of the protocol.

We can then proceed with the most important property of a key-exchange protocol: confidentiality

```
1 lemma Confidentiality :
2   "(All alice bob key #t1 #t2 .
3     SecretEstablished(alice, bob, key) @ #t1
4     #> &
5     SecretConfirmed(alice, bob, key) @ #t2 &
6     not (Ex #t3 . SecretReveal(key) @ #t3)
7     ==>
8     not (Ex #t3 . KU(key) @ #t3))"
```

The previous lemma states that if there has been no leakage of shared secrets, Alice and Bob can safely establish a key without the attacker deducing it from the previous or subsequent messages of the network. Again, this lemma is quickly proved without the help of human intervention.

The attack In 1981 [DS81], Denning and Sacco found a vulnerability in the protocol: if an attacker E manages to compromise an old value of Sk , it could easily replay the message $E \rightarrow B : \{Sk, A\}_{K_B}$ by impersonating Alice and thus forcing the two parties to communicate with a useless key.

It's easy to find a valid attack trace within our model:

```
1 lemma Attack :
2   exists-trace
3   "(Ex alice bob key #t #t1 .
4     SecretConfirmed(alice, bob, key) @ #t &
5     SecretConfirmed(alice, bob, key) @ #t1 &
6     #t < #t1 &
7     not (Ex #t2 .
8       SecretEstablished(bob, alice, key) @
9       #t2 & #t < #t2))"
```

In the same paper in which the authors explained the issue, they also proposed a simple solution to such problem: incorporating timestamps. The server (which we can assume is not malicious because otherwise we would always find trivial attacks since it is responsible to distribute the

keys) includes temporal information in the message sent to Alice: $S \rightarrow A : \{N_A, B, Sk, \{Sk, A\}_{K_B}\}_{K_A}$ becomes $S \rightarrow A : \{N_A, B, Sk, \{Sk, \text{timestamp}, A\}_{K_B}\}_{K_A}$ and, as a consequence, Alice sends to Bob the encrypted message $A \rightarrow B : \{Sk, \text{timestamp}, A\}_{K_B}$. Bob will thus be able to check whether the timestamp is recent or not and, assuming a decent time synchronization between the parties and that the attacker is not able to compromise the password instantly, will discard replayed messages. The fixed exchange is summarized in table ??

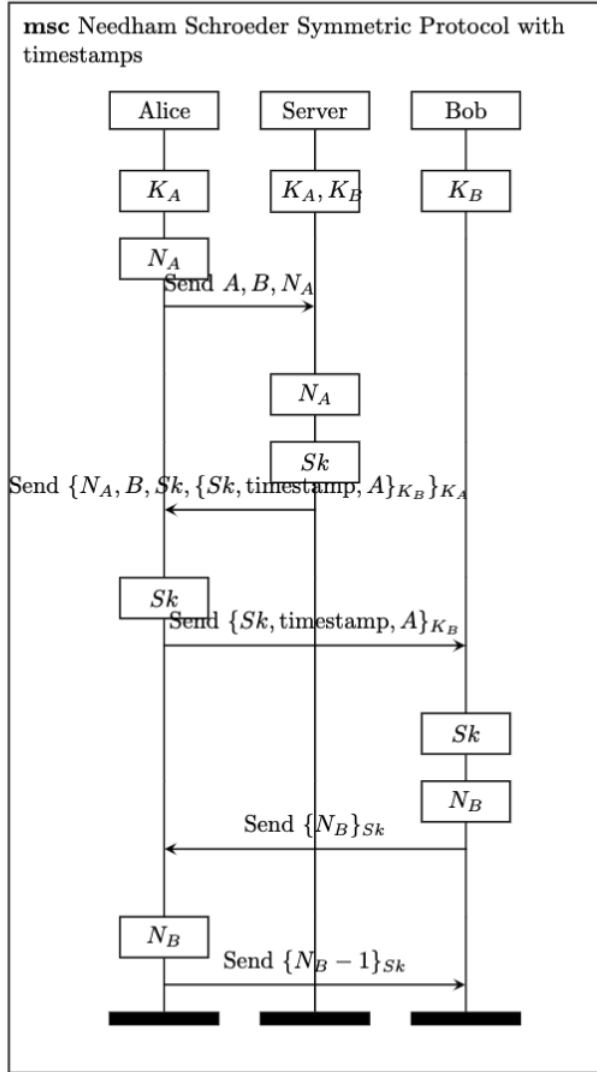


Figure 4. Needham-Schroeder protocol with added timestamps

To model this version of the protocol we must change the rules that formalize the intermediate exchanges to include such timestamps:

```

1 rule Server_answers_request :
2   let
3     message_to_bob = senc(<~sk, ~timestamp,
4       ↪ alice>, kb)
5     message_to_alice = senc(<nonce, bob, ~sk
6       ↪ , message_to_bob>, ka)
  
```

```

5   in
6   [
7     !Server(alice, ka),
8     !Client(alice, ka),
9     !Client(bob, kb),
10    In(<alice, bob, nonce>),
11    Fr(~sk),
12    Fr(~timestamp)
13  ]
14  --[ ServerSendsToAlice(message_to_alice),
15    Secret(~sk) ]->
16  [
17    Out(message_to_alice)
18  ]

1 rule Alice_receives_answer :
2   let
3     message_to_bob = senc(<sk1, timestamp,
4       ↪ alice>, k)
5     message_to_alice = senc(<nonce, bob, sk,
6       ↪ message_to_bob>, ka)
7   in
8   [
9     !Client(alice, ka),
10    !Client(bob, kb),
11    AliceQueriesServer(alice, bob, nonce),
12    In(message_to_alice)
13  ]
14  --[ AliceReceivesAnswer(message_to_alice,
15    ↪ nonce, message_to_bob, sk, ka),
16    AliceSendsToBob(message_to_bob),
17    SecretEstablished(alice, bob, sk) ]->
18  [
19    ExchangeInitiated(alice, bob, sk),
20    Out(message_to_bob)
21  ]

1 rule Bob_responds_to_Alice :
2   let
3     message_from_alice = senc(<sk, timestamp
4       ↪ , alice>, kb)
5     encrypted_nonce = senc(~nonce, sk)
6   in
7   [
8     In(message_from_alice),
9     Fr(~nonce),
10    !Client(alice, ka),
11    !Client(bob, kb),
12    !Server(bob, kb)
13  ]
14  --[ BobReceivesFromAlice(message_from_alice,
15    ↪ sk, kb),
16    SecretEstablished(bob, alice, sk,
17    ↪ timestamp),
18    BobSendsNonce(encrypted_nonce) ]->
19  [
20    SessionPending(bob, alice, ~nonce, sk),
21    Out(encrypted_nonce)
22  ]
  
```

We also must introduce a new restriction that models the checking of the timestamp by Bob:

```

1 restriction CheckingTimestamps :
2   "All alice bob sk timestamp #t1 #t2 .
3     SecretEstablishedWithTimestamp(bob,
4     ↪ alice, sk, timestamp) @ #t1 &
  
```

```

4      SecretEstablishedWithTimestamp(bob,
    ↪  alice, sk, timestamp) @ #t2
5      ==> #t1 = #t2"

```

We can see now that Tamarin when run on the `Attack` lemma is not able now to provide any valid trace, thus confirming that this attacks was solved.

3.1.3 Performance metrics

By running Tamarin on the files that contain the theories formalizing the protocol in the two forms, we will obtain the evaluation of the formulas modeling the security properties. Assuming that we want to analyze the theory of the unfixed Needham-Schroeder protocol contained in `./NS.spthy`, we will obtain the following output:

```

>>> tamarin-prover NS.spthy

=====
summary of summaries:

analyzed: NS.spthy

types (all-traces): verified (45 steps)
Sanity (exists-trace): verified (15 steps)
Confidentiality (all-traces): verified (64
  ↪ steps)
Attack (exists-trace): verified (18 steps)

=====

```

Similarly, by analyzing the formalization of the fixed protocol contained within file `./NS_fixed.spthy`, the tool provides the following result:

```

>>> tamarin-prover NS_fixed.spthy

=====
summary of summaries:

analyzed: NS_fixed.spthy

types (all-traces): verified (130 steps)
Sanity (exists-trace): verified (15 steps)
Confidentiality (all-traces): verified (19
  ↪ steps)
Attack (exists-trace): falsified - no trace
  ↪ found (294 steps)

=====

```

In order to gain some insight on the tool's performance, we can measure the time and space resources required by a series of executions of the prover on the single lemmas and calculate some statistics on it. Automating such procedure is pretty straightforward in any modern programming language, and in our case we chose to use Python along to GNU's time command line utility [GNU]:

```

import subprocess, re, json
from statistics import mean, stdev

def runTamarin(filename, lemmaName):
    command = f'gtime tamarin-prover {filename}.
    ↪ spthy --prove={lemmaName}'

```

```

process = subprocess.run(command.split(),
    ↪ stdout=subprocess.PIPE, stderr=subprocess.
    ↪ PIPE, universal_newlines=True)
gtimeResults = process.stderr.split('\n')
    ↪ [-3].split()
time = list(map(int, re.split('\:|\.',
    ↪ gtimeResults[2][:-7])))
milliseconds = (time[0] * 60 * 1000) + (time
    ↪ [1] * 1000) + (time[2] * 10)
maxMemory = int(gtimeResults[-1][:-13])
return milliseconds, maxMemory

```

```

FILES = ['NS', 'NS_fixed']
LEMMAS = ['types', 'Sanity', 'Confidentiality',
    ↪ 'Attack']
N_ITERATIONS = 100

results = {}

for file in FILES:
    for lemma in LEMMAS:
        times = []
        memories = []
        for i in range(N_ITERATIONS):
            print(f'Lemma: {lemma} | Iteration:
    ↪ {i+1}/{N_ITERATIONS}')
            newTime, newMemory = runTamarin(file
    ↪ , LEMMAS[0])
            times.append(newTime)
            memories.append(newMemory)
            results[lemma] = {}
            results[lemma]['maxTime'] = max(times)
            results[lemma]['maxMemory'] = max(
    ↪ memories)
            results[lemma]['minTime'] = min(times)
            results[lemma]['minMemory'] = min(
    ↪ memories)
            results[lemma]['avgTime'] = mean(times)
            results[lemma]['avgMemory'] = mean(
    ↪ memories)
            results[lemma]['stdevTime'] = stdev(
    ↪ times)
            results[lemma]['stdevMemory'] = stdev(
    ↪ memories)
        with open(f'benchmarks_{file}.json', 'w') as
    ↪ f:
            json.dump(results, f)

```

The script described above measures the resources used to prove each lemma during 100 different iterations. Note that each iteration requires a precomputation-phase overhead, which could be easily amortized by computing all the lemmas in the same execution. The results of our experimentation as described in tables 1 and 2.

As we can see, with this toy-protocol example, Tamarin is able to prove all the properties in the blink of an eye. Of course, when considering more complex formalizations with additional rules and bigger equational theories involved (such as the `diffie-hellman` or the `xor` built-ins), the proving process becomes more time-consuming (and often non-terminating). For example, in the reference GitHub repository [D'A23] for this paper it is also possible to find a Tamarin formalization of the X3DH exchange proposed by Marlinspike and Perrin [MP16] that, despite its brevity, manages to get the

Lemma	# of steps	Mean	Duration [ms]			Mean	Peak RAM usage [kB]		
			Max	Min	Deviation		Max	Min	Dev
types	123	697	770	660	15.09	46697	51248	44976	129
Sanity	15	701	800	670	18.20	46973	50208	44016	131
Confidentiality	98	706	770	660	14.75	46957	50224	43984	135
Attack	19	745	800	710	16.60	46702	51232	44992	134

Table 1. Tamarin’s performances during the verification of the Needham Schoroeder Symmetric protocol

Lemma	# of steps	Mean	Duration [ms]			Mean	Peak RAM usage [kB]		
			Max	Min	Deviation		Max	Min	Dev
types	130	912	1000	850	37.11	61134	63504	58352	87
Sanity	15	935	990	870	24.51	61155	62528	58288	91
Confidentiality	19	953	1040	900	28.58	61132	62496	59360	76
Attack	294	989	1510	890	86.09	61195	64528	59376	78

Table 2. Tamarin’s performances during the verification of the Needham Schoroeder Symmetric protocol after fixing

prover stuck during the precomputation phase due to the high number of Diffie Hellman exchanges required in the protocol.

4. Conclusions

In this essay we managed to present the Tamarin prover with a top-down approach: starting from a general introduction to the formal checking context background (in particular the symbolic model for protocol verification), we introduced the famous theorem prover by quickly explaining its syntax and usage. In order to provide a more concrete point of view on the matter, we also presented the Needham Schroeder Symmetric protocol and formalized its message exchange in Tamarin semantics. Then, we used the tool to prove some of its security properties and showed how it could be subject to a simple replay attack. After showing a potential fix to the problem, we proved that the proposed modification did, in fact, eliminated the attack. Finally, we provided some performance metrics on the tool.

The topic of automated formal verification of protocols is currently an area of active research within the symbolic artificial intelligence field. As computer scientists, this allows us to go through a less cumbersome verification process when checking protocols for correctness, while as users this provides us a better sense of trust with regards to the security features of the application we are using, since their exchanges are more easily formalized and verified against attackers.

References

- [AC04] A. Armando and L. Compagna. Satmc: A sat-based model checker for security protocols. In *Logics in Artificial Intelligence*, volume 3229, pages 730–733. Springer Berlin Heidelberg, Sep 2004.
- [BBB⁺19] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno. Sok: Computer-aided cryptography. Cryptology ePrint Archive, Paper 2019/1393, 2019.
- [BDS15] D. Basin, J. Dreier, and R. Sasse. Automated symbolic proofs of observational equivalence. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1144–1155, Oct 2015.
- [Bla16] B. Blanchet. Modeling and verifying security protocols with the applied pi calculus and proverif. *Foundations and Trends in Privacy and Security*, 1:1–135, Oct 2016.
- [BMV05] D. Basin, S. Mödersheim, and L. Vigano. Ofmc: A symbolic model checker for security protocols. *International Journal of Information Security*, 4:181–208, Jan 2005.
- [Bru12] B. 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.
- [CD09] V. Cortier and S. Delaune. A method for proving observational equivalence. In *2009 22nd IEEE Computer Security Foundations Symposium*, pages 266–276, Jul 2009.
- [CDD20] V. Cortier, S. Delaune, and J. Dreier. *Automatic Generation of Sources Lemmas in Tamarin: Towards Automatic Proofs of Security Protocols*, pages 3–22. Springer International Publishing, Sep 2020.
- [D'A23] D. D'Ambrosi. Symbolic verification of security protocols with tamarin. <https://github.com/dambrosidenis/Computer-Network-Security>, May 2023. Accessed: 2023-05-05.
- [DHRS18] J. Dreier, L. Hirschi, S. Radomirovic, and R. Sasse. Automated unbounded verification of stateful cryptographic protocols with exclusive or. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 359–373, Jul 2018.
- [DLM04] N Durgin, P. Lincoln, and J. Mitchell. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12:247–311, Feb 2004.
- [DS81] D. Denning and G. Sacco. Timestamps in key distribution protocols. *Commun. ACM*, 24:533–536, Aug 1981.
- [DY83] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
- [GNU] GNU. time(1) - linux man page. <https://linux.die.net/man/1/time>.
- [Mei13] S. Meier. *Advancing automated security protocol verification*. PhD thesis, ETH, 2013.
- [MP16] M. Marlinspike and T. Perrin. The x3dh key agreement protocol. *Open Whisper Systems*, 283:10, 2016.
- [MSCB13] S. Meier, B. Schmidt, C. Cremers, and D. Basin. The tamarin prover for the symbolic analysis of security protocols. In *Computer Aided Verification*, volume 8044, pages 696–701, Jul 2013.
- [NS78] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, Dec 1978.
- [Sch12] B. Schmidt. *Formal analysis of key exchange protocols and physical protocols*. PhD thesis, ETH, 2012.
- [SMCB12] B. Schmidt, S. Meier, C. Cremers, and D. Basin. Automated analysis of diffie-hellman protocols and advanced security properties. In *2012 IEEE 25th*

Computer Security Foundations Symposium, pages 78–94, 2012.

- [SS89] M. Schmidt-Schauss. Unification in permutative equational theories is undecidable. *Journal of Symbolic Computation*, 8(4):415–421, 1989.
- [Tea13] The Tamarin Team. Tamarin prover. <https://tamarin-prover.github.io>, 2013. Accessed: 2023-05-05.
- [Tea22] The Tamarin Team. *Tamarin-Prover Manual: Security Protocol Analysis in the Symbolic Model*, 2022.
- [Tur06] M. Turuani. The cl-atse protocol analyser. In *Term Rewriting and Applications, Lecture Notes in Computer Science*, volume 4098, pages 277–286, Aug 2006.
- [Vit21] N. Vitacolonna. Symbolic verification of security protocols with tamarin. <http://users.dimi.uniud.it/~angelo.montanari/SymbolicVerificationWithTamarin.pdf>, May 2021. Accessed: 2022-07-30.