Automated Symbolic Verification of the Needham-Schroeder Protocol

Denis D'Ambrosi[1]

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 allows 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

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

Contents

Introduction

> Security protocols are three-line programs that people still manage to get wrong
>
> Roger M. Needham

As computing becomes more obiquitous 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 Dolev Yao model

When dealing with the verification of internet protocols, two (main) paradigms are used to formally prove the security properties of both new and established message exchanges: the computational and the symbolic models. The first, initially introduced by Goldwasser, Micali [GM84], Rivest [GMR88], Yao [Yao82] and others, treats messages as bitstrings, cryptographic primitives as endomorphisms mapping objects in the space of bitstrings and the adversary as a probabilistic Turing machine. Within this model, given a security parameter (such as an encryption key) and a property (formula) that needs to be verified, the adversary must find a polynomial-time algorithm with respect to the size of the parameter that makes the formula false. If the probability of such procedure to work is negligible, then the property is considered as proved. This model manages to resemble real-world cryptography pretty closely, but specifing and verifying its theories can easily become a tedious and lengthy process.

At the cost of accuracy, the symbolic model (also known as the Dolev-Yao model [DY83]) abstracts from real-world cryptography operations by substituting actual crypto primitives with term-algebras. Encryption schemes are easily formalized through as binary isomorphisms: for example, given a secret key k and the relevant pair of cryptographic encryption and decryption function symbols senc_k, sdec_k and 1, symmetrical cryptography is defined through the following identity:

$$\text{sdec}_k \ \text{senc}_k = 1 \tag{1}$$

Only the entities that are in possess of k are able to encrypt or decrypt any message with it [Bru12]: this hypothesis is known as *perfect cryptography*.

Similarly to Equation 1, asymmetric encryption is modeled through the following identity:

$$\text{adec}_{pr} \ \text{aenc}_{pub} = \text{adec}_{pub} \ \text{aenc}_{pr} = 1 \tag{2}$$

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.

Given a message M and its image $\text{senc}_k \ M$ (or, $\text{aenc}_{pub} \ M$), we assume that it is impossible for an attacker who does not know k (or pr) to:

- guess or bruteforce k (or pr);
- manipulate $\text{senc}_k \ M$ (or $\text{aenc}_{pub} \ M$)
- infer any information about M from $\text{senc}_k \ M$ (or $\text{aenc}_{pub} \ M$).

This set of hypotheses represents both the strength and weakness of this paradigm: from a modeling point of view, the abstractions introduced effectlively strip down protocols leaving only their cryptographic primitives, thus allowing for far easier formalization and verification processes. Long and complex message exchanges between multiple parties are manageable within this model and, as we will soon show, various provers have been proposed with this purpose during recent years. On the other hand, by simplyfing cryptography down to only algebraic symbols, this model does not take into consideration neither possible bad implementation choices nor real-world attacks regarding the cryptographic primitives actually used. The computational model, by being less abstract, clearly covers a wider spectrum of risk scenarios, but with an non-negligible added complexity and still without being considered a completely inclusive paradigm (we can trivially think about side channel attacks, which are not being included in any threat model by definition).

Coming back to the Dolev-Yao model, a protocol is formalized as a series of algebraic 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 (but always following the perfect cryptography constraints described). After formalizing a protocol, its security goals can be expressed as:

- 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.

1.1 Computational 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]. To prove security goals, these softwares have to compute the set of terms that an attacker is able to deduce from the network while (possibly) multiple runs the protocol are executing; since such set can possibly be unbounded, the general problem can easily become undecidable and suffer of infinite state space: 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 [EG83].

In particular, as shown by N. Durgin et al. [DLM04], by restricting different sources we can ensure to reduce the model checking problem 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 unbounded 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].

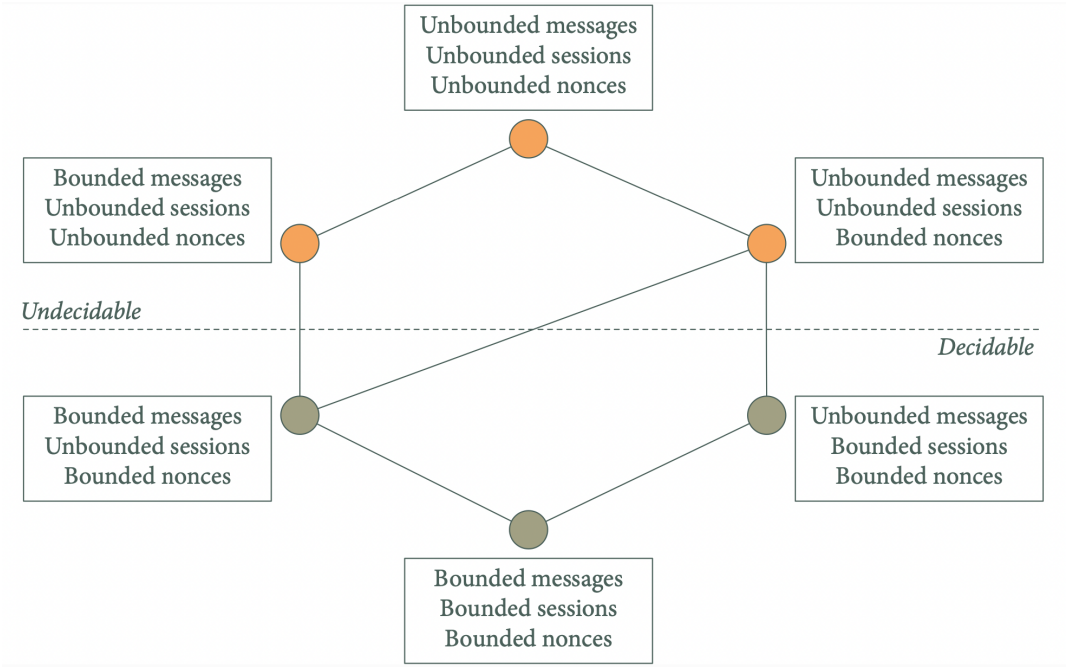

Figure 1. Decidability in symbolic verification. Image taken by N. Vitacolonna's presentation [Vit21]

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, as we will see in Section 2.6.3), returning inconclusive results (for example, ProverIf [Bla16] may return an invalid attack trace) or even allowing non-termination.

2. Tamarin Prover Overview

In 2012, researchers at ETH introduced a new symbolic verification software, the Tamarin prover. Such tool provides an automated search for attacks, advanced protocol analysis techniques, and an interactive theorem proving approach that enables security experts to manually guide the proof process. 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 solve any doubts about its use.

2.1 Term algebra

As explained in Section 1, the Dolev-Yao model formalizes cryptographic messages within a term algebra; thus, in order to introduce Tamarin's syntax, we firsly have to take a look at (some) of the theoretical foundations behind the tool.

Definition 2.1: Signature

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

Signatures are fundamental, since are the building blocks of our theory. As we will soon see, all the algebraic terms we introduced before (for example in Equations 1 and 2) will be expressed as function symbols.

Definition 2.2: Σ-terms

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

- $\chi \subseteq \mathscr{T}_\Sigma(\chi)$
- $t_1,...,t_n \in \mathscr{T}_\Sigma(\chi) \land f/n \in \Sigma \implies f(t_1,...,t_n) \in \mathscr{T}_\Sigma(\chi)$

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

While in this introduction we will deal with domain-agnostic logical terms, in practice Tamarin discriminates between 3 different types of terms to model various assumptions on different pieces of knowledge:

- *Fresh* (groud) terms are prefixed by a ~ and model information that we can assume a party can check for freshness.
- *Public* names are juxtaposed by a $ and formalize ground information that any party within the network knows.
- *Normal* terms are any form of term: fresh, public or built upon function symbols.

Σ-terms allow us to recursively define algebraic terms starting from ground istances of variables. In real life we are able to combine multiple cryptographic primitives in order to build protocol messages: analogously, this set is made up of all the (countably infinite) possible combinations of function applications starting from ground terms.

Clearly, we need now a way to "compare" messages: as we previously explained in Equations 1 and 2, the decryption of the encryption of any message (of course with the correct keys) must match the message itself. Similarly, all the primitives we will introduce must have some mechanism to check whether two terms belonging to a given set $\mathscr{T}_\Sigma(\chi)$ match each other or not.

Definition 2.3: Substitution

Given a signature Σ and a set of variables χ, with $\Sigma \cap \chi = \varnothing$, a substitution is a function $\sigma : \chi \to \mathscr{T}_\Sigma(\chi)$.

In our case, we do not only want to be able to build (and compare) terms upon ground instances of variables, but also build them upon other terms. As a consequence, we must generalize the definition of substitution:

Definition 2.4: Mapping

Given a function $f/n \in \Sigma$, a mapping $\sigma' : \mathscr{T}_\Sigma(\chi) \to \mathscr{T}_\Sigma(\chi)$ is an extension of a substitution $\sigma : \chi \to \mathscr{T}_\Sigma(\chi)$ such that

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

As we will soon see, mappings will be crucial for meaningful comparisons between terms. Of course, in our case we will narrow our focus on only sensible mappings that allow us to correctly model the cryptographic primitives needed. Each primitive will have a set of rules that will define its behaviour and such set will be formalized through equations:

Definition 2.5: Equation over Σ

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

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 $\mathscr{T}_\Sigma(\chi)/_{=_E}$: two terms $t,u \in \chi \cup \mathscr{T}_\Sigma(\chi)$ are equal in modulo E if and only if they belong to the same class:

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

We will soon see how the use of equational theories enables us to formalize cryptographic primitives, but, before continuing, it is sensible to briefly talk about a decidability issue that regards this aspect of the term algebra.

Definition 2.6: (Σ,E)-Unification

Given a signature Σ, a set of variables χ, with $\Sigma \cap \chi = \varnothing$ and an equational theory E, two terms $t,u \in \mathscr{T}_\Sigma(\chi)$ 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]: to overcome this issue Tamarin recommends users to only define *subterm convergent theories* to ensure termination of the unification process [CR12]. In practice, this constraint is never enforced by the tool, thus users must be careful when introducing custom theories (notice that exploiting only Tamarin's default equations overcomes this issue but may not be sufficient to model all the possible scenarios).

Definition 2.7: Convergent Theory

Before 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

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

2.2 Equational theories in Tamarin

By default, Tamarin includes the function symbol `*/2`, used in the *AC-equational theory*:

$$x*(y*z) \simeq (x*y)*z \text{ (associativity)}$$
$$x*y \simeq y*z \text{ (commutativity)}$$

This theory clearly models some properties of multiplication. Similarly, the tool provides function symbols (`pair/2`, `fst/1` and `snd/1`) and equations to work with pairs:

$$fst(pair(x,y)) \simeq x$$
$$snd(pair(x,y)) \simeq y$$

Other function symbols are shipped within the tool, but must be loaded into a theory through the `built-ins` directive:

hashing The `hashing` theory defines the symbol `h/1` and no equations. This clearly models the one-wayness of an ideal hash function, which cannot be reversed (the only way to determine whether the pre-image of `h(x)` is `x` is to find `x`).

symmetric encryption The `symmetric-encryption` theory models Equation 1 in a very natural way: after providing symbols `senc/2` and `sdec/2`, it defines the equation

$$sdec(senc(m,k),k) = m$$

which enforces the fact that the only way to access an encrypted term is to know the decryption key (perfect cryptography assumptions).

asymmetric encryption Similarly to last paragraph, the `asymmetric-encryption` theory formalizes Equation 2 through symbols `pk/1`, `aenc/2` and `adec/2` and the equation

$$adec(aenc(m,pk(sk)),sk) = m$$

By not providing other equations, this ensures that an attacker is not able neither to decrypt an encrypted message without the private secret, nor to deduce a private key from it corrispondent public counterpart.

signing The `signing` theory allows to model digital signing schemes. It introduces symbols `sign/2`, `verify/3`, `pk/1`, `true/0` and the equation

$$verify(sign(m,sk),m,pk(sk)) = true$$

Yet again, not providing other equations enforces the fact that the attacker is not able to forge valid signatures without first finding a private key.

diffie hellman Perhaps the most complex built-in theory is the `diffie-hellman` one: it defines function symbols `inv/1`, `1/0`, `^/2` and `*/2` and the following equations:

$$(x^y)^z \simeq x^{(y*z)}$$
$$x^1 \simeq x$$
$$x*y \simeq y*x$$
$$(x*y)*z \simeq x*(y*z)$$
$$x*1 \simeq x$$
$$x*inv(x) \simeq 1$$

Note that this theory allows to effectively model Diffie Hellman exchanges (both classical and with elliptic curves) while providing the remaining multiplication properties not captured by the AC theory (namely the inverse and the identity).

Other built-in theories There are other built-in theories that can be included within a Tamarin model:

1. `revealing-signing`
2. `bilinear-pairing`
3. `xor`
4. `multiset`
5. `reliable-channel`

for the sake of brevity we will not introduce them, but we encourage any interested reader into looking at their definitions within the manual [Tea22].

User-defined theories Furthermore, a user can define additional custom theories to model any cryptographic primitive or mechanism required for a security protocol.

For example, one may want to model the homomorphic properties of a symmetric encryption scheme (for example vanilla RSA's invariance under multiplication): to do so, it is sufficient to add the following equation to a formalization after the `equations` keyword (of course assuming the relative built-in theories have already been included):

$$aenc(m1,k)*aenc(m2,k) \simeq aenc(m1*m2,k)$$

Similarly, when dealing with elliptic curve cryptography, it is possible to use the same keypair both for the Ed25519 signing schema and the X25519 exchange procedure [Tho21]. Such behaviour can be modeled by introducing the following equation [1] (again, considering the `diffie-hellman` and `asymmetric-encryption` built-ins as already included):

$$g^x \simeq pk(x)$$

where g is a public constant symbolizing the generator of the curve.

Finally, it has to be noted that a custom symbol can be defined with the `[private]` keyword to prevent the attacker from creating new terms with it. This could be useful if, for example, we wanted to model a protocol in which there is a cryptographic primitive (like a secret hash function) that, for some reason, is accessible only by the intended parties of an exchange. It is well know that "security by obscurity" is not a principle to follow at all, but this attribute makes the formalization of similar contexts straightforward. All the other functions are considered public and thus can be employed by the user and the attacker alike.

[1] Note that this is not entirely true: name clashing on symbols may force the user to rename some functions for this to work. In any case, it is possible to simply avoid including the built-ins and explicitly defining the relative equational theories following the definitions introduced in this section without using reserved names.

2.3 Formalizing protocols as sets of rewriting rules

In Tamarin, the execution of a protocol is modeled by the evolution of a multiset of facts. In knowledge representation, facts are "true" predicates that have a fixed arity and are composed of terms; Tamarin follows this definition and requires the user to define them with a starting capital letter and provides two different versions of them:

- *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 syntatically prefixed by an exclamation mark).

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

Definition 2.9: Labelled rewriting rule

Given a multiset $\Gamma_t = \{F_0, ..., F_n\}$ and a sequence of multisets $trace_t = \langle a_0, ..., a_{t-1} \rangle$ at a time t, we can define a rewrite rule as a triple of multisets $RR = \langle L, A, R \rangle$ (written as $RR = L - [A] \to R$) such that:

- we can apply RR to Γ_t if there is at least one ground instance (i.e. an instance with no variables) $rr = l - [a] \to r$ 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

$$\Gamma_{t+1} = \Gamma_t \setminus^{\#} lin(l) \cup^{\#} r$$
$$trace_{t+1} = \langle a_0, ..., a_{t-1}, a \rangle$$

in which $\setminus^{\#}$ and $\cup^{\#}$ are the multiset equivalent operations for set difference and union and $lin(l)$ is the multiset of linear facts belonging to l (notice that persistent facts are never removed from the state). 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 each rule is labelled by a name N, thus can be seen as a pair (N, RR)

Notice the detail about the trace being a sequence and not a set: since security properties will be later on specified as guarded fragments of first order temporal logic on set of traces, the ordering relation present in a sequence allows to define temporal relations between different action facts.

In particular, given a set of rewriting rules P, Tamarin will analyze the set of all the possible traces generated by executions of P:

Definition 2.10: Set of possible traces

Given a set of labelled rewriting rules $P = \{(N_1, RR_1), ..., (N_m, RR_m)\}$, we define the set of possible traces generated by P as

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

where A_i is RR_i's action facts multiset.

Furthermore, it is important to note that security properties will be specified on specific types of traces, namely *observable traces*.

Definition 2.11: Observable trace

Given a trace tr, we can compute its relative observable trace tr_{obs} by removing all the empty multisets from it:

$$tr_{obs} = \langle A_i | A_i \in tr \wedge A_i \neq \varnothing^{\#} \rangle$$

To better understand how such trace (both observable and non) is derived from a protocol's execution, we can take a look at the following example rewriting system:

$$P = \{(N_1, \varnothing^{\#} - [\{\{\text{Init}(0)\}\}] \to \{\{A(0)\}\}),$$
$$(N_2, \{\{A(x)\}\} - [\varnothing^{\#}] \to \{\{B(x)\}\})$$
$$(N_3, \{\{B(x)\}\} - [\{\{\text{Concl}(x)\}\}] \to \varnothing^{\#})\}$$

Let us assume we are starting with an empty state Γ_0 and apply rules N_1, N_2, N_1, N_3, in this order. The state would evolve as:

$$\Gamma_0 = \varnothing^{\#} \setminus^{\#} \varnothing^{\#} \cup^{\#} \{\{A(0)\}\} = \{\{A(0)\}\}$$
$$\Gamma_1 = \{\{A(0)\}\} \setminus^{\#} \{\{A(0)\}\} \cup^{\#} \{\{B(0)\}\} = \{\{B(0)\}\}$$
$$\Gamma_2 = \{\{B(0)\}\} \setminus^{\#} \varnothing^{\#} \cup^{\#} \{\{A(0)\}\} = \{\{A(0), B(0)\}\}$$
$$\Gamma_3 = \{\{A(0), B(0)\}\} \setminus^{\#} \{\{B(0)\}\} \cup^{\#} \varnothing^{\#} = \{\{A(0)\}\}$$

while the generated trace would be

$$tr = \langle \{\{\text{Init}(0)\}\}, \{\{\varnothing^{\#}\}\}, \{\{\text{Init}(0)\}\}, \{\{\text{Concl}(0)\}\}, \rangle$$

and the observable trace

$$tr_{obs} = \langle \{\{\text{Init}(0)\}\}, \{\{\text{Init}(0)\}\}, \{\{\text{Concl}(0)\}\} \rangle$$

2.3.1 Dolev Yao rules

Although the above described multiset rule rewriting system can be used for general model checking, Tamarin provides a set of built-in rules to model connections within Dolev-Yao's adversary controlled network:

- `[] --[]-> [ Fr(~msg) ]` allows for the generation of new fresh values.
- `[ Fr(~msg) ] --[]-> [ K(~msg) ]` allows for the generation of new fresh values by the attacker.

- `[ Out(msg) ] --[]-> [ K(msg) ]` allows the attacker to eavesdrop all messages travelling through the network.
- `[ K(msg) ] --[ K(msg) ]-> [ In(msg) ]` allows user to retrieve messages from the attacker-controlled network.
- `[] --[]-> [ K($x) ]` allows the attacker to discover all public names.
- `[ K(x1,...,xn) ] --[]-> [ K(f(x1,...,xn)) ]` allows the attacker to apply n-ary functions to arguments he already knows.

Notice that these rule allow us to easily specify security properties about the knowledge of the attacker (for example, confidentiality): if we wanted to ensure that a protocol does not reveal a certain secret `sec`, we just need to require that `K(sec)` does not appear within any of the multisets of the observable trace.

Tamarin only provides built-in facts for communication over insecure and reliable channels, but if we needed to formalize other types of connections we could use the above rule as a blueprint to define them. For example, to model a confidential channel (thus a connection in which the attacker could send, but not read from messages), one may define the following rules:

```
rule Send_Over_Confidential_Channel :
    [ ConfOut(msg) ]
    --[]->
    [ ConfIn(msg) ]

rule Attacker_Sends_Over_ConfChannel :
    [ K(msg) ]
    --[]->
    [ ConfIn(msg) ]
```

As we can see, there is no rule that allows the adversary to learn anything from the confidential channel, but he might send any forged message through it. On the other hand, an honest user could employ the `ConfOut` and `ConfIn` facts to model sending and retrieval on the channel. Possibly, if we wanted, we could also differentiate between channels by augmenting the rules with a connection identifier:

```
rule Send_Over_Confidential_Channel :
    [ ConfOut(msg, channel) ]
    --[]->
    [ ConfIn(msg, channel) ]

rule Attacker_Sends_Over_ConfChannel :
    [ K(msg), K(channel) ]
    --[]->
    [ ConfIn(msg, channel) ]
```

Similarly, to model an authentic channel (a connection where integrity, but not confidentiality is guaranteed), we could define the following rule:

```
rule Send_Over_Authentic_Channel :
    [ AuthOut(msg) ]
    --[ K(msg) ]->
    [ AuthIn(msg), K(msg) ]
```

By not including a rule that allows the attacker to produce `AuthOut` or `AuthIn` facts, we ensure that such channel cannot be polluted with forged messages. Again, we could trivially differentiate between channels or build both confidential and authentic (secure) connections with analogous rules.

2.4 Trace properties

As previously anticipated, in Tamarin security properties are expressed as guarded fragments of first order logic. The keyword 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 verifing the formula, or `all-traces`, which are proved by negating the property and trying to find a counterexample. To specify a lemma, we can combinate the following atoms:

- false $\bot$;
- 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 2.12: Correctness

Given 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:

$$P \vDash \phi \iff traces(\phi) \subseteq traces(P)$$

On the contrary, if $P \nvDash \phi$, all traces belonging to $traces(P)/traces(\phi)$ 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 report, 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 2.13: 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 2.14: 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 during 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 infine loop of computation or not by monitoring wheter recursive structures of terms are being produced or not. Furthermore, by guiding the proof, the user can easily guess how to build an effective oracle to speed up lenghty 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.

2.6.4 Restrictions

Similarly to lemmas, *restrictions* are specified through guarded fragments of first order logic. The difference between the two

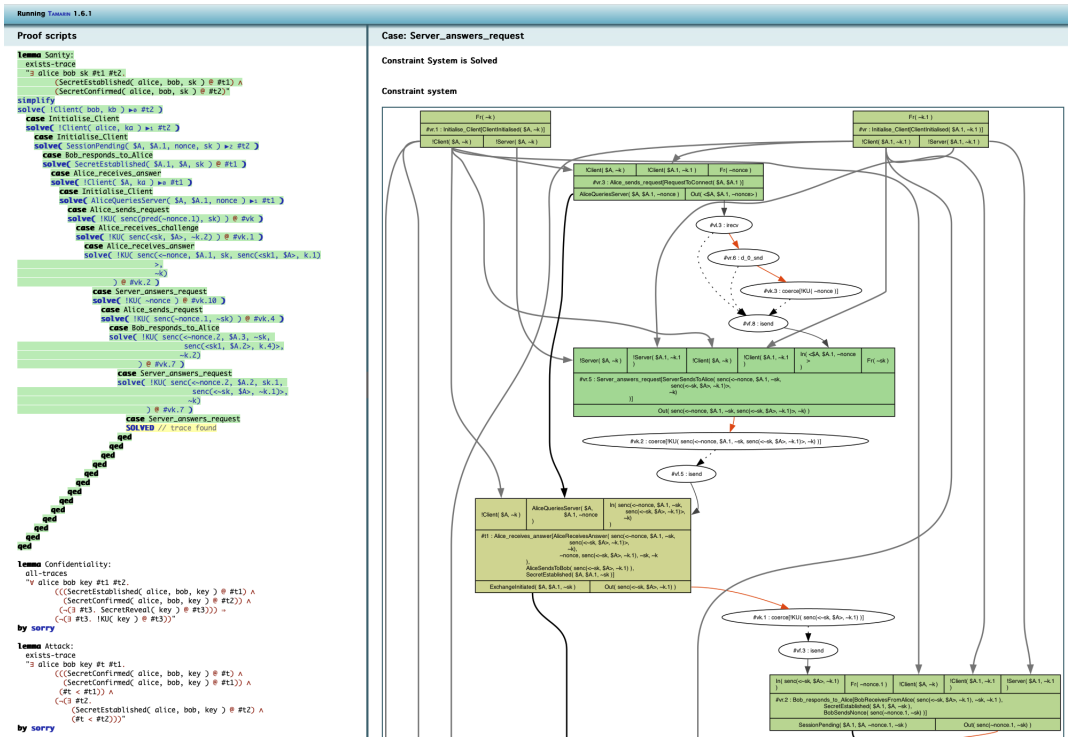

Figure 2. Tamarin's interactive mode

is that while lemmas express properties to prove, a restriction limits the possible traces of a protocol to the executions that satisfy the specified formula. An example of restriction use could be to avoid the application of the same rule twice:

```
rule Generic_rule :
    [ OldFact(x) ]
    --[ OnlyOnce() ]->
    [ NewFact(x) ]

restriction Do_not_repeat :
    "All #i #j .
        OnlyOnce() @ #i &
        OnlyOnce() @ #j
            ==> #i = #j"
```

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), 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 Figure 3.

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.

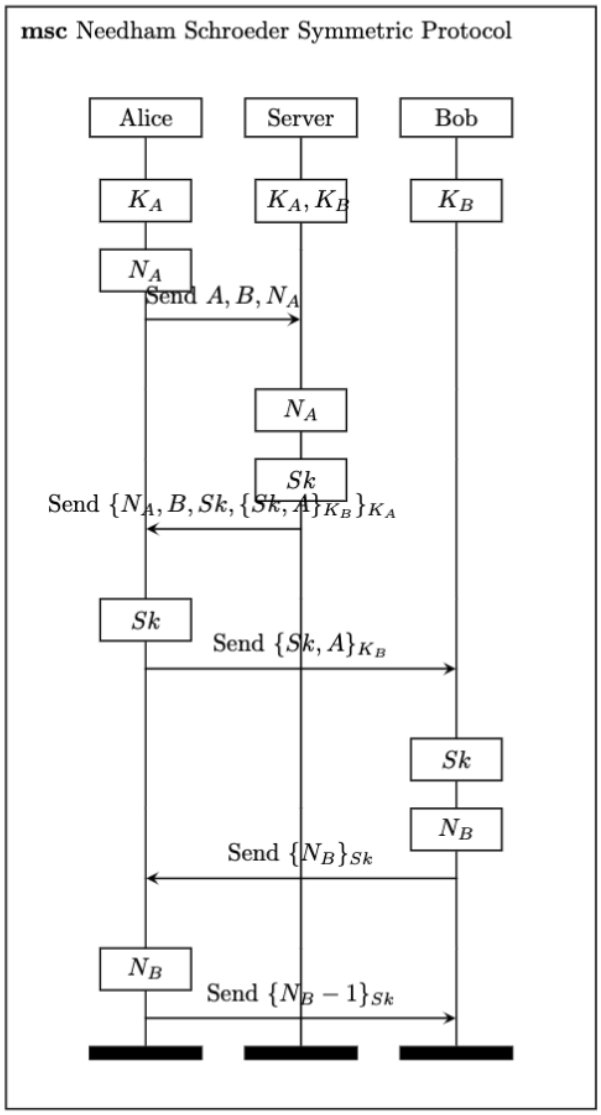

Figure 3. Needham-Schroeder Symmetric protocol

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:

```
rule Initialise_Client :
    [
        Fr(~k)
    ]
    --[ ClientInitialised($A, ~k) ]->
    [
        !Client($A, ~k),
        !Server($A, ~k$)
    ]
```

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:

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

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

```
rule Alice_sends_request :
    [
        !Client(alice, ka),
        !Client(bob, kb),
        Fr(~nonce)
    ]
    --[ RequestToConnect(alice, bob) ]->
    [
        AliceQueriesServer(alice, bob, ~nonce),
        Out(<alice, bob, ~nonce>)
    ]

restriction NoSelfCommunication:
    "All client #t .
        RequestToConnect(client, client) @ #t
    ↪ ==> 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.

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

```
rule Server_answers_request :
    let
        message_to_bob = senc(<~sk, alice>, kb)
        message_to_alice = senc(<nonce, bob, ~sk
    ↪ , message_to_bob>, ka)
    in
    [
        !Server(alice, ka),
        !Server(bob, kb),
        !Client(alice, ka),
        !Client(bob, kb),
        In(<alice, bob, nonce>),
        Fr(~sk)
    ]
    --[ ServerSendsToAlice(message_to_alice) ]->
    [
        Out(message_to_alice)
    ]
```

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.

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

```
rule Alice_receives_answer :
    let
        message_to_alice = senc(<nonce, bob, sk,
    ↪  message_to_bob>, ka)
    in
    [
        !Client(alice, ka),
        AliceQueriesServer(alice, bob, nonce),
        In(message_to_alice)
    ]
    --[ AliceReceivesAnswer(message_to_alice,
    ↪ nonce, message_to_bob, sk, ka),
        AliceSendsToBob(message_to_bob),
        SecretEstablished(alice, bob, sk) ]->
    [
        ExchangeInitiated(alice, bob, sk),
        Out(message_to_bob)
    ]
```

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.

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

```
rule Bob_responds_to_Alice :
    let
        message_from_alice = senc(<sk, alice>,
    ↪ kb)
        encrypted_nonce = senc(~nonce, sk)
    in
    [
        In(message_from_alice),
        Fr(~nonce),
        !Client(alice, ka),
        !Client(bob, kb),
        !Server(bob, kb)
    ]
    --[ BobReceivesFromAlice(message_from_alice,
    ↪  sk, kb),
        SecretEstablished(bob, alice, sk),
        BobSendsNonce(encrypted_nonce) ]->
    [
        SessionPending(bob, alice, ~nonce, sk),
        Out(encrypted_nonce)
    ]
```

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.

$A \to B : \{N_B - 1\}$ 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.

```
function: pred/1

rule Alice_receives_challenge :
    let
        nonce_received = senc(nonce, sk)
        nonce_sent = senc(pred(nonce), sk)
    in
    [
        !Client(alice, ka),
        ExchangeInitiated(alice, bob, sk),
        !Client(bob, kb),
        In(nonce_received)
    ]
```

```
    --[ AliceReceivesNonce(nonce_received, nonce
    ↪ , sk) ,
        AliceSendsNonce(nonce_sent)]->
    [
        !Session(alice, bob, sk),
        Out(nonce_sent)
    ]
```

Again, if we modeled both the predecessor and the successor function, the theory would become indecidable 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:

```
rule Bob_receives_challenge_answer :
    let
        received_nonce = senc(pred(nonce), sk)
    in
    [
        !Client(bob, kb),
        !Client(alice, ka),
        SessionPending(bob, alice, nonce, sk),
        In(received_nonce)
    ]
    --[ BobReceivesNonce(received_nonce, nonce,
    ↪ sk),
        SecretConfirmed(alice, bob, sk) ]->
    [
        !Session(bob, alice, sk)
    ]
```

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 specify 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:

```
rule Secret_reveal :
    [ !Session(client1, client2, sk) ]
    --[ SecretReveal(sk) ]->
    [ 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:

```
lemma types [sources] :
    "(All #t sk kb message .
        BobReceivesFromAlice(message, sk, kb) @
    ↪ #t ==>
        (Ex #t1 . AliceSendsToBob(message) @ #t1
    ↪ )
        | (Ex #t1 . KU(sk) @ #t1 & KU(kb) @ #t1)
    ↪ )
    &
    (All #t sk ka message nonce message_bob .
        AliceReceivesAnswer(message, nonce,
    ↪ message_bob, sk, ka) @ #t ==>
        (Ex #t1 . ServerSendsToAlice(message) @
    ↪ #t1)
        | (Ex #t1 . KU(sk) @ #t1 & KU(ka) @ #t1
            & KU(nonce) @ #t1 & KU(message_bob)
    ↪ @ #t1))
    &
    (All #t sk encnonce nonce .
        AliceReceivesNonce(encnonce, nonce, sk)
    ↪ @ #t ==>
        (Ex #t1 . BobSendsNonce(encnonce) @ #t1)
        | (Ex #t1 . KU(sk) @ #t1 & KU(nonce) @ #
    ↪ t1)
        | (Ex #t1 . KU(encnonce) @ #t1))
    &
    (All #t sk encnonce nonce .
        BobReceivesNonce(encnonce, nonce, sk) @
    ↪ #t ==>
        (Ex #t1 . AliceSendsNonce(encnonce) @ #
    ↪ t1)
        | (Ex #t1 . KU(sk) @ #t1 & KU(nonce) @ #
    ↪ t1)
        | (Ex #t1 . KU(encnonce) @ #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:

```
restriction NoSendingPrivateKeys :
    "All client key #t .
        ClientInitialised(client, key) @ #t
        ==> 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 wheter our formalization of the protocol allows for a successful run of the exchange. This can be verified through and `exists-trace` lemma:

```
lemma Sanity :
    exists-trace
    "Ex alice bob sk #t1 #t2 .
        SecretEstablished(alice, bob, sk) @ #t1
    ↪ &
```

```
        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

```
lemma Confidentiality :
    "(All alice bob key #t1 #t2 .
        SecretEstablished(alice, bob, key) @ #t1
    ↪ &
        SecretConfirmed(alice, bob, key) @ #t2 &
        not(Ex #t3 . SecretReveal(key) @ #t3)
        ==>
            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 and 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:

```
lemma Attack :
    exists-trace
    "Ex alice bob key #t #t1 .
        SecretConfirmed(alice, bob, key) @ #t &
        SecretConfirmed(alice, bob, key) @ #t1 &
        #t < #t1 &
        not(Ex #t2 .
            SecretEstablished(bob, alice, key) @
    ↪ #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 4

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

```
rule Server_answers_request :
    let
        message_to_bob = senc(<~sk, ~timestamp,
    ↪ alice>, kb)
        message_to_alice = senc(<nonce, bob, ~sk
    ↪ , message_to_bob>, ka)
    in
    [
        !Server(alice, ka),
        !Client(alice, ka),
        !Client(bob, kb),
        In(<alice, bob, nonce>),
        Fr(~sk),
        Fr(~timestamp)
    ]
    --[ ServerSendsToAlice(message_to_alice),
        Secret(~sk) ]->
    [
        Out(message_to_alice)
    ]
```

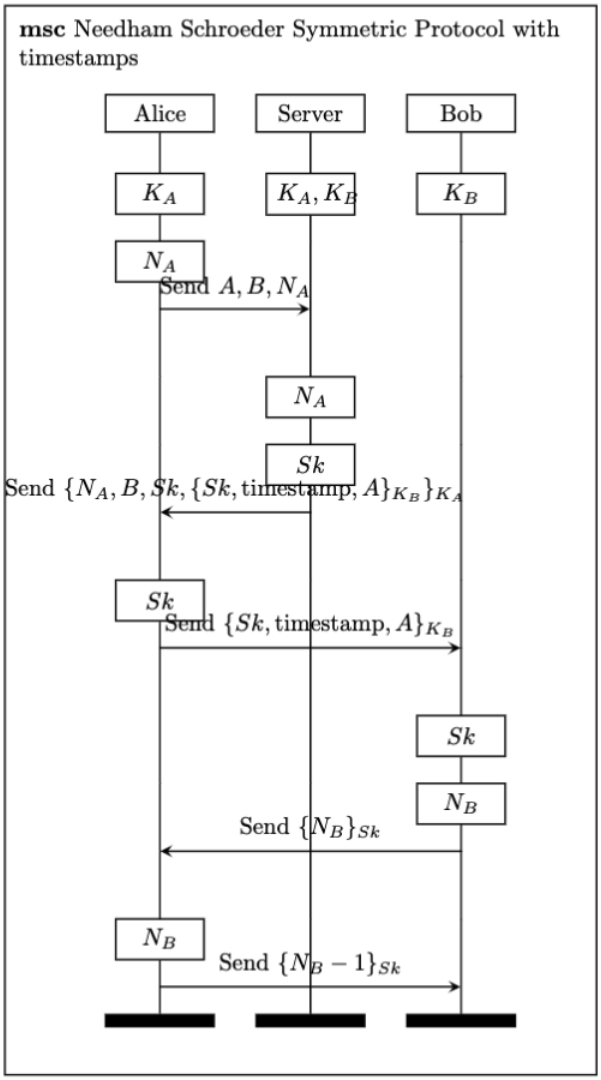

Figure 4. Needham-Schroeder protocol with added timestamps

```
rule Alice_receives_answer :
    let
        message_to_alice = senc(<nonce, bob, sk,
    ↪ message_to_bob>, ka)
    in
    [
```

```
        !Client(alice, ka),
        !Client(bob, kb),
        AliceQueriesServer(alice, bob, nonce),
        In(message_to_alice)
    ]
    --[ AliceReceivesAnswer(message_to_alice,
    ↪ nonce, message_to_bob, sk, ka),
        AliceSendsToBob(message_to_bob),
        SecretEstablished(alice, bob, sk) ]->
    [
        ExchangeInitiated(alice, bob, sk),
        Out(message_to_bob)
    ]

rule Bob_responds_to_Alice :
    let
        message_from_alice = senc(<sk, timestamp
    ↪ , alice>, kb)
        encrypted_nonce = senc(~nonce, sk)
    in
    [
        In(message_from_alice),
        Fr(~nonce),
        !Client(alice, ka),
        !Client(bob, kb),
        !Server(bob, kb)
    ]
    --[ BobReceivesFromAlice(message_from_alice,
    ↪  sk, kb),
        SecretEstablished(bob, alice, sk,
    ↪ timestamp),
        BobSendsNonce(encrypted_nonce) ]->
    [
        SessionPending(bob, alice, ~nonce, sk),
        Out(encrypted_nonce)
    ]
```

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

```
restriction CheckingTimestamps :
    "All alice bob sk timestamp #t1 #t2 .
        SecretEstablishedWithTimestamp(bob,
    ↪ alice, sk, timestamp) @ #t1 &
        SecretEstablishedWithTimestamp(bob,
    ↪ alice, sk, timestamp) @ #t2
            ==> #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 Perfomance 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 --prove

==============================================
    ↪
summary of summaries:

analyzed: NS.spthy

  processing time: 7.66s

  types (all-traces): falsified - found trace
    ↪ (36 steps)
  Sanity (exists-trace): verified (14 steps)
  Confidentiality (all-traces): verified (48
    ↪ 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 --prove

==============================================
    ↪
summary of summaries:

analyzed: NS_fixed.spthy

  processing time: 6.34s

  types (all-traces): verified (684 steps)
  Sanity (exists-trace): verified (14 steps)
  Confidentiality (all-traces): verified (16
    ↪ steps)
  Attack (exists-trace): falsified - no trace
    ↪ found (55 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)
    res = process.stderr.split('\n')[-3].split()
    time = list(map(
        int,
        res.split('\:|\.', res[2][:-7])))
    milliseconds = (time[0] * 60 * 1000) +
        (time[1] * 1000) +
        (time[2] * 10)
    maxMemory = int(res[-1][:-13])
    return milliseconds, maxMemory
```

		Duration [ms]				Peak RAM usage [kB]			
Lemma	# of steps	Mean	Max	Min	Deviation	Mean	Max	Min	Deviation
types	123	697	770	660	15.09	46697	51248	44976	1295.62
Sanity	15	701	800	670	18.20	46973	50208	44016	1316.64
Confidentiality	98	706	770	660	14.75	46957	50224	43984	1357.73
Attack	19	745	800	710	16.60	46702	51232	44992	1340.96

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

		Duration [ms]				Peak RAM usage [kB]			
Lemma	# of steps	Mean	Max	Min	Deviation	Mean	Max	Min	Deviation
types	130	912	1000	850	37.11	61134	63504	58352	875.15
Sanity	15	935	990	870	24.51	61155	62528	58288	912.24
Confidentiality	19	953	1040	900	28.58	61132	62496	59360	761.72
Attack	294	989	1510	890	86.09	61195	64528	59376	786.25

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

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

out = {}
for file in FILES :
    for lemma in LEMMAS :
        times = []
        mems = []
        for i in range(N_ITERATIONS) :
            res = runTamarin(file, LEMMAS[0])
            times.append(res[0])
            mems.append(res[1])
        out[lemma] = {}
        out[lemma]['maxTime'] = max(times)
        out[lemma]['maxMemory'] = max(mems)
        out[lemma]['minTime'] = min(times)
        out[lemma]['minMemory'] = min(mems)
        out[lemma]['avgTime'] = mean(times)
        out[lemma]['avgMemory'] = mean(mems)
        out[lemma]['stdevTime'] = stdev(times)
        out[lemma]['stdevMemory'] = stdev(mems)
    with open(f'output_{file}.json', 'w') as f:
        json.dump(out, 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 are 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 [MP16] that, despite its brevity, manages to get the prover stuck during the precomputation phase due to the high number of Diffie Hellman exchanges required by 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 verification background, we introduced the actual tool 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.

[CR12] Y. Chevalier and M. Rusinowitch. Decidability of equivalence of symbolic derivations. *Journal of Automated Reasoning*, 48(2):263–292, 2012.

[D'A23] D. D'Ambrosi. Needham-schroeder symmetric key protocol automated analysis. `https://github.com/dambrosidenis/NeedhamSchroeder_Automated_Analysis`, 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.

[EG83] 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.

[GM84] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[GMR88] S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[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.

[Tho21] E. Thormarker. On using the same key pair for ed25519 and an x25519 based kem. Cryptology ePrint Archive, Paper 2021/509, 2021.

[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.

[Yao82] A. C. Yao. Theory and application of trapdoor functions. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 80–91, 1982.