Formalizing Social Engineering attacks in the Symbolic Model

July 24, 2023

Contents

Introduction

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

Today's digital infrastructure is increasingly reliant on cryptographic protocols in order to function correctly and efficiently while mitigating the risks of intrusion by malicious parties. Online applications are overtaking traditional physical offices in the name of efficiency; but while mimicking the often decades-old tasks themselves (like purchasing public transportation tickets or voting) in the cyberspace is generally not a big deal on itself, guaranteeing the security of these virtual processes against emerging digital threats demands a lot more research effort. Formal verification tools allow us to construct checkable models of cryptographic protocols and to verify them against well-defined threat models to ensure that such exchanges do not hide any flaws in their design, however, they often rely on strong assumptions regarding the caution and knowledge of the involved users. In this paper, we want to introduce a possible solution for a more complete analysis of protocols which heavily rely on partecipants' input based on the ideas of human knowledge and human error.

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 (mainly from a user's perspective), while in the following section 3 we will show how we can formalize Social Engineering attacks using said tool's syntax in a very intuitive way. Finally, in section 4 we will summarize the contents 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 already established message exchanges: the *computational* and the *symbolic model*. 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 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:

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

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

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

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

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 $\mathrm{senc}_k \ M$ (or, $\mathrm{aenc}_{pub} \ M$), we assume that it is impossible for an attacker who does not know k (or pr) to:

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

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 of their low-level implementation complexity, keeping only the required cryptographic primitives, and thus allowing for far easier formalization and verification processes. Long and complex message exchanges between multiple parties are generally more 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 terms, 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 in the symbolic model, multiple automatic tools were proposed [BBB$^+$19]. In order to verify 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 infinite, the general task can quickly become undecidable and suffer of the infinite state space problem: we could have an unbounded number of sessions for each protocol, each one with an unlimited amount of nonces and recursive messages with no base case. Without bounding at least some of these possible 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 of infinity we can ensure to reduce the model checking problem to different complexity classes; for example if we limited the number of nonces and messages, we would end up with a DEXP-complete problem, while if we restricted the number of sessions, the problem will still be affected from unbounded state space, but its solution will be NP-complete.

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 2.5.3), returning inconclusive results (for example, ProverIf [Bla16] may return an invalid attack trace) or, more generally, 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 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 is formalism.

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 $\mathcal{T}_\Sigma(\chi)$ as the minimal set such that

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

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 $\mathcal{T}_\Sigma(\chi)$ 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 $\sigma : \chi \to \mathcal{T}_\Sigma(\chi)$.

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 $\sigma' : \mathcal{T}_\Sigma(\chi) \to \mathcal{T}_\Sigma(\chi)$ is an extension of a substitution $\sigma : \chi \to \mathcal{T}_\Sigma(\chi)$ 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 $t, u \in \mathcal{T}_\Sigma(\chi)$. 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 $\mathcal{T}_\Sigma(\chi)/_{=_E}$: 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:

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

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 issues that regard 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 $t, u \in \mathcal{T}_\Sigma(\chi)$ are (Σ, E)-unifiable if there is at least a mapping σ such that $t\sigma =_E u\sigma$.

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.

Normally, unification modulo theories is undecidable [SS89]: to overcome this issue Tamarin recommends users to only define *convergent theories* with finite variant property to ensure termination of the unification process [DDKS17]. Formally defining the finite variant property is out of the scope for this paper, but, intuitively we can express this property as follows: given a generic term t, we can compute all of its normal forms of its possible istances in advance. Before the publication of the 2017 paper, which provided new decidability results for the unification problem, Tamarin required its users to only define *subterm convergent equational theories*; this constraint has been recently lifted, although the manual [Tea22] still recommends to express custom theories as subterm convergent when possible for efficiency reasons.

As of this moment, Tamarin offers 10 different built-in equational theories that users can directly include in their formalization to straightforwardly model cryptographic operations. To provide some intuitive examples, we wanted to show the equations for pair construction and destruction and symmetric encryption:

$$\texttt{fst}(\texttt{pair}(x, y)) = x \tag{3}$$

$$\texttt{snd}(\texttt{pair}(x, y)) = y \tag{4}$$

$$\texttt{sdec}(\texttt{senc}(m, k), k) = m \tag{5}$$

Figure 1: Equations belonging to the default and `symmetric-encryption` equational theories

Note that equations 3 and 4 rely on the signature $\Sigma_{\text{default}} = \{\texttt{pair}/2, \texttt{fst}/1, \texttt{snd}/2\}$, while equation 5 (which closely models the identity introduced at 1) relies on $\Sigma_{\text{sym-enc}} = \{\texttt{senc}/2, \texttt{sdec}/2\}$.

Overall, the theories provided by the tool cover most of the cryptographic primitives adopted in classical protocol specification, but, if a user required a functionality that was not provided out-of-the-box, it would be easy for him to include additional function symbols and the relative equations through the `functions` and `equations` keywords.

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.

2.2 Formalizing protocols as sets of rewriting rules

We have seen how cryptographic operations can be formalized in Tamarin's syntax. Communication protocols involve far more complex dynamics than can not be expressed by atomic term manipulations, thus we introduce the idea of *facts*.

In Tamarin, the execution of a protocol is modeled by the evolution of a multiset[1] of facts (which represent the state of the system at any given moment). In knowledge representation, facts are "true" predicates that have a fixed arity and are composed of terms belonging to the supporting term-algebra; Tamarin follows this simple definition and requires the user to define them with a starting capital letter. In particular, there are two different versions of this construct that can be exploited during protocol modeling:

- *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 of facts $\Gamma_t = \{\{F_0, ..., F_n\}\}$ 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 $RR = L \xrightarrow{A} R$) such that:

- we can apply RR to Γ_t if there is at least one ground instance (i.e. an instance with no variables) $rr = l \xrightarrow{a} r$ of RR so that $l \subseteq^{\#} \Gamma_t$

[1] Within this paper we define multisets through the $\{\{...\}\}$ notation: an empty multiset is represented as $\{\{\}\}$, while a multiset containing elements x, y, z is represented as $\{\{x, y, z\}\}$

- applying rr to Γ_t yields to a new state Γ_{t+1} and an increased trace $trace_{t+1}$ obtained as

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

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.

Although we introduced general rewriting rules, we still need to restrict our focus on rules that follow Dolev Yao's model semantics: as a consequence, we can formally define a protocol as a set of rules which follow some precise constraints:

Definition 2.10: Protocol rules

A protocol rule is a multiset rewriting rule $l \xrightarrow{a} r$ such that

1. l, a, r do not contain fresh names
2. l does not contain `K` and `Out` facts
3. r does not contain `Fr` and `K` facts
4. The argument of any `Fr` fact belongs to set of fresh terms
5. r does not contain the function symbol $*$
6. $l \xrightarrow{a} r$ satisfies:
   - $vars(r) \subseteq vars(l) \cup \mathcal{V}_{\text{pub}}$
   - l only contains irreducible function symbols from the given signature or it is an instance if a rule that satisfies both conditions

After defining the idea of protocol within our rule rewriting system, it is clearly necessary to introduce also a formal definition of the possible executions of said protocol:

Definition 2.11: Set of possible traces

Given a set of labelled protocol rewriting rules P, we define the set of possible traces generated by it as

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

where A_i is the action fact of the i^{th} rule applied. Note that the fact `Fresh` will be soon introduced formally, but for now it is enough to know that it is used to generate new terms.

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

Definition 2.12: Observable trace

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

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

To better understand how such trace (both observable and not) is derived from a protocol's execution, we can take a look at the following example rewriting system:

$$\begin{aligned} P = \{ & (N_1, [\,] \xrightarrow{\texttt{Init}(0)} \{\{A(0)\}\}), \\ & (N_2, \{\{A(x)\}\} \rightarrow \{\{B(x)\}\}) \\ & (N_3, \{\{B(x)\}\} \xrightarrow{\texttt{Concl}(x)} [\,]) \} \end{aligned}$$

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:

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

while the generated trace would be

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

and the observable trace

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

2.2.1 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:

- $[\,] \xrightarrow{\texttt{Fresh}(\ msg)} [\texttt{Fr}(\ msg)]$ allows for the generation of new fresh values.
- $[\texttt{Fr}(\ msg)] \rightarrow [\texttt{K}(\ msg)]$ allows for the generation of new fresh values by the attacker (expressed by the `K` fact)
- $[\texttt{Out}_{\text{ins}}(msg)] \rightarrow [\texttt{K}(msg)]$ allows the attacker to eavesdrop all messages travelling through the network.
- $[\texttt{K}(msg)] \xrightarrow{\texttt{K}(msg)} [\texttt{In}_{\text{ins}}(msg)]$ allows user to retrieve messages from the attacker-controlled network.
- $[\,] \rightarrow [\texttt{K}(\$x)]$ allows the attacker to discover all public names.
- $[\texttt{K}(x_1, ..., x_n)] \rightarrow [\texttt{K}(\texttt{f}(x_1, ..., x_n))]$ 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 possible observable traces derived from said protocol execution.

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 rules 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:

$$[\texttt{Out}_{\text{conf}}(msg)] \rightarrow [\texttt{In}_{\text{conf}}(x)]$$
$$[\texttt{K}(msg)] \rightarrow [\texttt{In}_{\text{conf}}(x)]$$

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:

$$[\texttt{Out}_{\langle \text{conf}, channel \rangle}(msg, channel)] \rightarrow [\texttt{In}_{\langle \text{conf}, channel \rangle}(msg, channel)]$$
$$[\texttt{K}(msg), \texttt{K}(channel)] \rightarrow [\texttt{In}_{\langle \text{conf}, channel \rangle}(msg, channel)]$$

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

$$[\texttt{Out}_{\text{auth}}(msg)] \xrightarrow{\texttt{K}(msg)} [\texttt{In}_{\text{auth}}(msg), \texttt{K}(msg)]$$

By not including a rule that allows the attacker to produce $\texttt{Out}_{\text{auth}}$ or $\texttt{In}_{\text{auth}}$ facts, we ensure that such channel cannot be polluted with forged messages. By introducing analogous rules, we could clearly model channels that are both authentic and confidential (*secure channels*), or differentiate between channels also in other types of communication mediums.

2.3 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 $\perp$;
- 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).

Some security properties are straightforward to formalize as formulas with this syntax and semantics: for example, assuming in our theory there is at least a rule that has an action fact $\texttt{Secret}(m)$ that indicates that a party supposes that m is confidential, secrecy can be expressed as:

$$\forall m, t_1 : Secret(m)@t_1 \Rightarrow \neg\exists t_2 : K(m)@t_2$$

Keeping in consideration that also properties can clearly define sets of traces similarly to protocol rules (given a formula ϕ, $traces(\phi)$ is the set of all traces that satisfy ϕ), we can define correctness as follows:

Definition 2.13: 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:

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

On the contrary, if $P \nvDash \phi$, all traces belonging to $traces(P)ntraces(\phi)$ 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].

2.4 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 2.14: Trace equivalence

Two 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 2.15: Diff equivalence

Two 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 (an oriented graph that describes the causal relations between the various applications of ground rules in a protocol execution) 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].

Clearly diving more deeply within this extension's background theory diverges from this paper's scope. On the other hand, looking at the matter with a more practical approach, we still wanted to point out that this additional functionality is crucial for formalizing indistinguishability properties, which are at the root of many privacy aspects.

2.5 Aiding termination

Since unbounded protocol verification in the symbolic model is an inherently undecidable problem, Tamarin provides some additional functionalities engineered to aid termination.

2.5.1 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.

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.

2.5.2 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.

2.5.3 Interactive mode

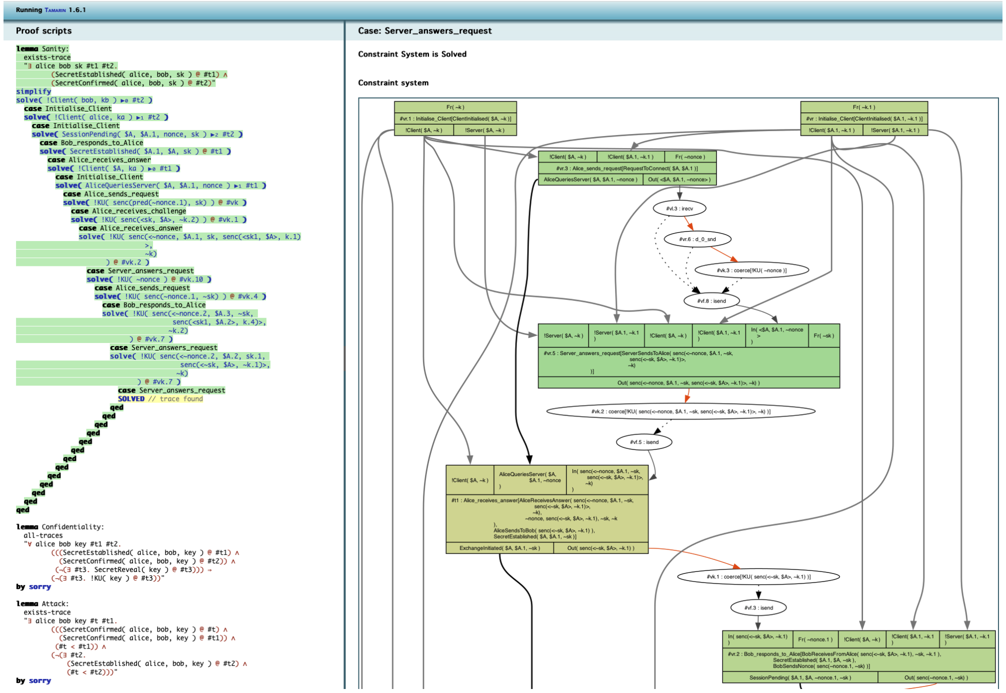
The Tamarin interactive mode is a powerful feature that enables users to guide the proof search process and interactively inspect the demonstration as it is being constructed. When using the interactive mode, users can provide hints and guidance to the Tamarin prover, which can significantly speed up 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 whether infinite 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.

2.5.4 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 satisfy the specified formula. An example of restriction use could be to avoid the application of the same rule twice:

$$[\texttt{OldFact}(x)] \xrightarrow{\texttt{OnlyOnce()}} [\texttt{NewFact}(x)]$$

$$\phi : \forall i, j : \texttt{OnlyOnce}()@i \land \texttt{OnlyOnce}@j \Rightarrow i = j$$



Constraint System is Solved

Constraint system

```
lemma Sanity:
  exists-trace
  "∃ alice bob sk #t1 #t2.
         (SecretEstablished( alice, bob, sk ) @ #t1) ∧
         (SecretConfirmed( alice, bob, sk ) @ #t2)"
simplify
solve( !Client( bob, kb ) ▶0 #t2 )
  case Initialise_Client
  solve( !Client( alice, ka ) ▶1 #t2 )
    case Initialise_Client
    solve( SessionPending( $A, $A.1, nonce, sk ) ▶2 #t2 )
      case Bob_responds_to_Alice
      solve( SecretEstablished( $A.1, $A, sk ) @ #t1 )
        case Alice_receives_answer
        solve( !Client( $A, ka ) ▶0 #t1 )
          case Initialise_Client
          solve( AliceQueriesServer( $A, $A.1, nonce ) ▶1 #t1 )
            case Alice_sends_request
            solve( !KU( senc(pred(~nonce.1), sk) ) @ #vk )
              case Alice_receives_challenge
              solve( !KU( senc(<sk, $A>, ~k.2) ) @ #vk.1 )
                case Alice_receives_answer
                solve( !KU( senc(<~nonce, $A.1, sk, senc(<sk1, $A>, k.1)
                                 >,
                                 ~k)
                       ) @ #vk.2 )
                  case Server_answers_request
                  solve( !KU( ~nonce ) @ #vk.10 )
                    case Alice_sends_request
                    solve( !KU( senc(~nonce.1, ~sk) ) @ #vk.4 )
                      case Bob_responds_to_Alice
                      solve( !KU( senc(<~nonce.2, $A.3, ~sk,
                                        senc(<sk1, $A.2>, k.4)>,
                                       ~k.2)
                             ) @ #vk.7 )
                        case Server_answers_request
                        solve( !KU( senc(<~nonce.2, $A.2, sk.1,
                                          senc(<~sk, $A>, ~k.1)>,
                                         ~k)
                               ) @ #vk.7 )
                          case Server_answers_request
                          SOLVED // trace found
                        qed
                      qed
                    qed
                  qed
                qed
              qed
            qed
          qed
        qed
      qed
    qed
  qed
qed

lemma Confidentiality:
  all-traces
  "∀ alice bob key #t1 #t2.
         (((SecretEstablished( alice, bob, key ) @ #t1) ∧
           (SecretConfirmed( alice, bob, key ) @ #t2)) ∧
          (¬(∃ #t3. SecretReveal( key ) @ #t3))) ⇒
         (¬(∃ #t3. !KU( key ) @ #t3))"
by sorry

lemma Attack:
  exists-trace
  "∃ alice bob key #t #t1.
         (((SecretConfirmed( alice, bob, key ) @ #t) ∧
           (SecretConfirmed( alice, bob, key ) @ #t1)) ∧
          (#t < #t1)) ∧
         (¬(∃ #t2.
             (SecretEstablished( alice, bob, key ) @ #t2) ∧
             (#t < #t2)))"
by sorry
```

Figure 2: Tamarin's interactive mode

2.5.5 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 Formalizing Social Engineering attacks

When verifying cryptographic protocols, we tend to abstract the parties involved in the exchanges to perfect machines which follow flawlessly the rules of the scheme under analysis. In practice, such operations are performed by humans, which are far for perfect and may introduce unexpected behaviours into the dynamics of a protocol (for example by falling for a social engineering attack), leading to possible catastrophical outcomes. In a recent report by Gitnux [Git23], the authors state that

> Social engineering is a prevalent threat, with 90% of data breaches having social engineering components and 62% of businesses experiencing attacks

highlighting the fact that human error is a key element in real-world threat analysis that must be taken into serious consideration when examining security-oriented protocols. Several types of cyberattacks have been developed in recent times to exploit such vulnerability: a comprehensive summary of this field of research is available at [SK19], from which we have taken Figure 3 for a quick reference.

To narrow the gap between the attacks normally considered in formal protocol verification and the threat surface of actual implementation of said schemes, Basin et al. proposed in 2016 [BRS16] an innovative solution to model erroneous deviations from protocol specification within the Dolev Yao model. In the rest of this section, we will introduce the ideas behind such article, underlining how simple it is to implement these ideas within Tamarin's fact rewriting mechanics. Note that the paper actually includes some results and notation from previous publications without prior introduction: to provide a more comprehensible explaination of the material we will include additional information also from articles [Low97] and [BRS15] and the appendixes of [BRS16], devoid of an explicit notice.

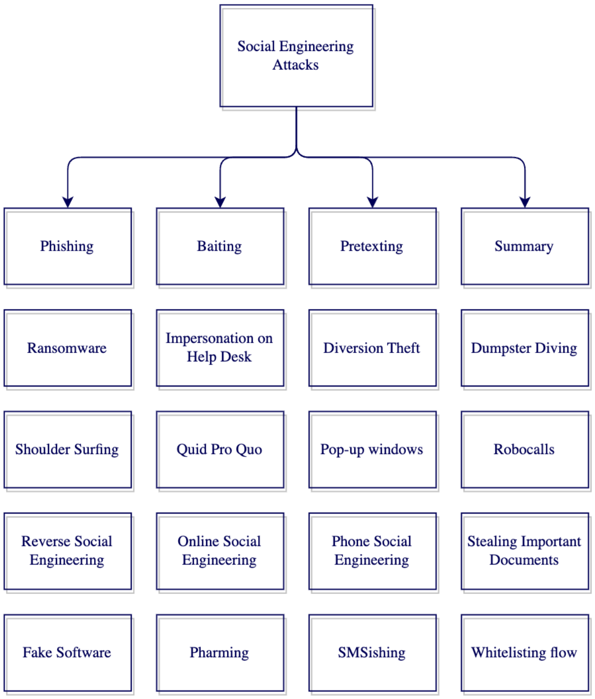


Figure 3: Summary of the categories of Social Engineering Attacks. Image inspired by Figure 4 of [SK19]

3.1 Including human knowledge in protocol formalization

Before defining errors in human behaviour, we need to actually establish which are the capabilities of said humans within our model. It is not unreasonable to presume that a person can perform simple operations but not complex computations unaided; in particular, we assume that a human can:

1. Send and receive messages on specified channels
2. Concatenate messages
3. Split concatenated messages

On the other hand, humans are not able to encrypt or decrypt messages unaided.

A key concept in this formalization is the concept of *human knowledge*: each individual partaking to an exchange stores a set of (tagged) information initially defined at initialisation and then updated through the retrieval of new messages from the network. To generalise this idea, the authors suggest to unbound the number of information storable by any human partecipant and to model actual knowledge evolution through a database of facts in the form of $!HK(H, \langle t, x \rangle)$, where:

- The fact is declared as persistent (through the juxtaposition of the exclamation mark) to portray that any information can be consumed infinitely many times
- H is the human agent taken into consideration
- t is the *tag* for the data stored
- x is the actual information known by H

The tag is required to denote the human's interpretation of messages: intuitively, we need to be able to distinguish between different types of information (passwords, emails, etc...), thus we have to define a recursive function that is able to associate a label to any term belonging to the term algebra $\mathcal{T}_\Sigma(X)$ we are taking into consideration. At first, we need to assume to have the additional injective function $t : \Sigma \cup X \rightarrow \mathcal{L}$ (where $\mathcal{L}$ is the alphabet of intermediate labels) which associates to any ground term and function symbol of our theory a unique, atomic tag. We can then use such function to define $T : \mathcal{T}_\Sigma(X) \rightarrow \mathcal{L}^*$ as:

$$T(m) = \begin{cases} t(m) & \text{if } m \in \Sigma \cup X \\ t(f) \cdot T(a_1) \cdot \ldots \cdot T(a_n) & \text{if } m = f(a_1, \ldots, a_n), f \text{ is a function symbol in } \Sigma \text{ and } a_i \in \mathcal{T}_\Sigma(X) \end{cases}$$

where $\cdot$ is the concatenation operator. Our objective is to map the set of labels to the set of public constants $\mathcal{C}_{\text{pub}}$ included in the algebra itself. Since both $\mathcal{L}^*$ and $\mathcal{C}_{\text{pub}}$ are countably infinite sets, there must be an injective mapping from the first set to the latter. By composing such function with T, we obtain the function $\mathit{preTag} : \mathcal{T}_\Sigma(X) \rightarrow \mathcal{C}_{\text{pub}}$. The last step in our construction is to define the actual labelling map $\mathit{tag} : \mathcal{T}_\Sigma X \rightarrow \mathcal{T}_{\mathcal{C}_{\text{pub}} \cup \{\text{pair}\}}$ by including the possibility of term concatenation:

$$\mathit{Tag}(m) = \begin{cases} \langle \mathit{Tag}(m_1), \mathit{Tag}(m_2) \rangle & \text{if } m = \langle m_1, m_2 \rangle \\ \mathit{preTag}(m) & \text{otherwise} \end{cases}$$

This additional step is done to allow humans to create new knowledge based on the concatenation of pre-existing information (remember that this operations is one of the few operations that individuals can perform unaided). The rules proposed by the authors require the use of such function work with the human-knowledge database facts `!HK`: a generic information x will be actually stored as `!HK`$(H, \langle \mathit{Tag}(x), x \rangle)$.

Before continuing, it is helpful to define the $\vdash_H$ predicate, defined over pairs of tag-message pairs:

$$\langle t, m \rangle \vdash_H \langle t', m' \rangle \iff \exists i, k : 1 \leq i \leq k \wedge t = \langle t_1, \ldots, t_k \rangle \wedge m = \langle m_1, \ldots, m_k \rangle \wedge t' = t_i \wedge m' = m_i$$

Intuitively, $\langle t, m \rangle \vdash_H \langle t', m' \rangle$ if and only if agent H can select $\langle t', m' \rangle$ from $\langle t, m \rangle$.

3.2 Formalizing human errors

From a global perspective, a cryptographic protocol is typically defined in some form of Alice and Bob notation. In particular, we are going to use an extended notation that differentiates the channels used for communication based the form of security they guarantee (in the following definitions, let us assume that we want to represent a channel between parties A and B):

- An *insecure channel*: $A{\circ\!\!\rightarrow\!\circ}B$
- An *authentic channel*: $A{\bullet\!\!\rightarrow\!\circ}B$
- A *confidential channel*: $A{\circ\!\!\rightarrow\!\bullet}B$
- A *secure channel*: $A{\bullet\!\!\rightarrow\!\bullet}B$

Do not forget that, although Tamarin only features insecure channels (the ones with the weakest assumptions) by default, but, as we showed in 2.2.1 it is really easy to define rules to include also other types of channels.

If, on the other hand, we want to consider the actions perfomed by any agent specifically, we can focus on *role scripts*. A role script for an agent H in a given protocol is the sequence of operations that H has to execute in order to successfully complete the run of the exchange on his side, thus we can see them as "projections" onto parties of entire protocol specifications. More formally, a role script is a sequence of *events* $e \in \mathcal{T}_{\Sigma \cup \text{RoleActions}}(X)$, where RoleActions $= \{\texttt{Send}, \texttt{Receive}, \texttt{Start}, \texttt{Fresh}\}$, where the top-level function of e belongs to RoleActions. Such functions have a well defined meaning:

- `Send`(A, l, P, m) means that agent A sends on a channel of type l the message m to the apparent party P
- `Receive`(A, l, P, m) means that agent A receives from a channel of type l a message with expected pattern m from the apparent party P

$$\begin{array}{ll} 0.\ S & : \mathrm{knows}(R) \\ 0.\ R & : \mathrm{knows}(S, m_2) \\ 1.\ S{\circ\!\!\rightarrow\!\circ}R & : \mathrm{fresh}(m_1).m_1 \\ 2.\ R{\bullet\!\!\rightarrow\!\bullet}S & : m_2 \end{array}$$

Figure 4: Toy protocol in Alice and Bob notation

$$\langle \mathtt{Start}(S, R), \mathtt{Fresh}(S, m_1), \mathtt{Send}(S, \mathrm{ins}, R, m_1), \mathtt{Receive}(S, \mathrm{sec}, R, m_2)\rangle$$

Figure 5: Role script of role S for the Toy protocol

- $\mathtt{Start}(A, t)$ indicates the initial knowledge of term t by agent A
- $\mathtt{Fresh}(A, t)$ indicates that agent A produces a fresh term t

A brilliant example of the projection of a protocol specification onto a role script is provided in the article:
In turn, any role script can be translated into a set of Tamarin rewriting rules that formalize the behaviour of the given party (in this case S) into the system:

$$\begin{gathered} [\ ] \xrightarrow{\mathtt{Start}(S,R)} [\mathtt{AgSt}(S, 0, R)] \\ [\mathtt{AgSt}(S, 0, R), \mathtt{Fr}(m_1)] \xrightarrow{\mathtt{Fresh}(S,m_1)} [\mathtt{AgSt}(S, 1, \langle R, m_1\rangle)] \\ [\mathtt{AgSt}(S, 1, \langle R, m_1\rangle)] \xrightarrow{\mathtt{Send}(S,\mathrm{ins},R,m_1)} [\mathtt{AgSt}(S, 2, \langle R, m_1\rangle), \mathtt{Out}_{\mathrm{ins}}(\langle S, R, m_1\rangle)] \\ [\mathtt{AgSt}(S, 2, \langle R, m_1\rangle), \mathtt{In}_{\mathrm{sec}}(\langle R, S, m_2\rangle)] \xrightarrow{\mathtt{Receive}(S,\mathrm{sec},R,m_2)} [\mathtt{AgSt}(S, 3, \langle R, m_1, m_2\rangle)] \end{gathered}$$

Assuming we have the corresponding rules also for R, we can see that a succesfull execution of the protocols that follows the specification of 3.2 will induce a trace that mirrors the role script of an agent (given that we ignore the action facts belonging to other parties' actions). Furthermore, note that the function symbols belonging to RoleActions are now used as action facts: this is natural since in Tamarin we can only specify formulas (within lemmas and restrictions) over facts and not directly over terms.

After formally introducing role scripts, we can define a human error as *any deviation of a human from his role script.* Any human that makes mistakes is said a *fallible human.* On the other hand, an agent which adheres perfectly to his specification is said an *infallible human.*

Normally, in a formal verification context we would take into consideration only infallible humans. Such assumption would be reasonable for (almost) completely automatic protocols that necessitate little to no input by users and thus present slim chances of uncorrect executions. Conversely, when dealing with interactive protocols which require a lot of user intervention (for example any 2-factor authentication protocol) it is imperative to include the possibility that an agent makes a mistake, either because of carelessness or as a consequence of fraudolent attacker behaviour.

Fallible humans can be introduced through two main approaches (summarized in Figure 6), that we will now briefly explain.

3.2.1 Skilled Human Approach

The *Skilled Human Approach* starts by firstly taking into consideration an infallible human that does not drift from a given protocol's specification and then by introducing other humans, which can make a fixed number of mistakes (called *skilled humans*). Each new agent introduced on the basis of another previous human induces a supersets of the latter's possible behaviours: in this approach, we weaken the infallible human until the security properties are not guaranteed anymmore. It is interesting to note that the data

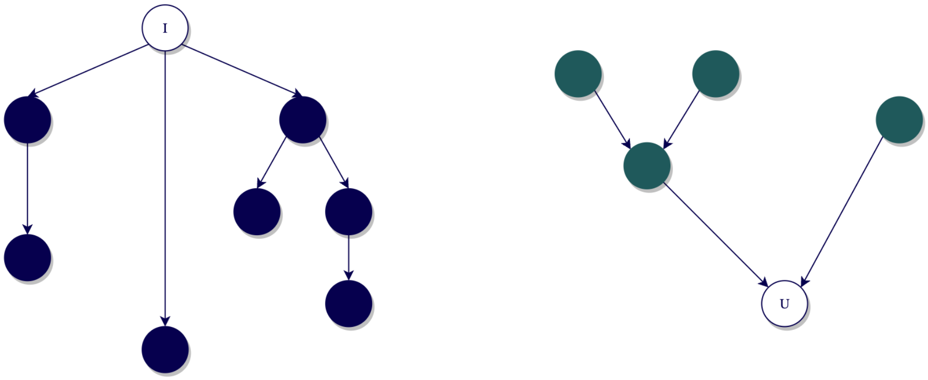


Figure 6: Approaches to introducing fallible humans. In the graphs, I is the infallible human, U is the untrained human and an edge from a node N to a node M denotes that N induces a subset of the behaviours of M. Blue nodes are *skilled humans* while teals nodes represent *trained humans*. Image inspired by Figure 1 of [BRS16]

structure that naturally depicts this approach is not a sequence, but instead a directed acyclic graph: we can consider different routes while weakening our actors (for example, by allowing the agents to fall for a phishing mail or to skip a QR code verification, but not necessarily both of them together), thus we can find some (possibly different) minimal sets of actions that must be executed rigorously according to the protocol in order to avoid security concerns.

3.2.2 Rule-Based Human Approach

The *Rule-Based Human Approach* begins with the introduction of an *untrained human*, which is capable of executing any possible behaviour (for example, leaking passwords without concern, skipping any additional verification step, communicating verification codes received out-of-band to the attacker, etc ...) and then strengthens such agent by imposing restrictions on the acceptable actions. This formalization models a human that does not know the protocol specification but adheres to some basic guidelines (for example, it is sensible to assume that a user does not know the dynamics of an given protocol, but he is aware that he must not communicate passwords outside of the authenticated plaforms).

Within Tamarin's fact rewriting system it is quite natural to formalize humans through the rule-based approach: firstly, we can model the untrained human behaviour through the following rules:

$$[\texttt{Fr}(x)] \xrightarrow{\texttt{Fresh}(H,\langle t,x\rangle)} [\texttt{!HK}(H,\langle t,x\rangle)] \tag{6}$$

$$[\texttt{!HK}(H,\langle t,x\rangle)] \xrightarrow{\texttt{Send}(H,\text{ins},P,\langle t,x\rangle)} [\texttt{Out}_{\text{ins}}(\langle H,P,\langle t,x\rangle\rangle)] \tag{7}$$

$$[\texttt{In}_{\text{ins}}(\langle P,H,\langle t,x\rangle\rangle)] \xrightarrow{\texttt{Receive}(H,\text{ins},P,\langle t,x\rangle)} [\texttt{!HK}(H,\langle t,x\rangle)] \tag{8}$$

These rules clearly follow the semantics introduced in Section 3.2 when explaining Role Scripts. We can trivially define rules equivalent to 7 and 8 for the remaining types of channels $(\text{auth}, \text{conf}, \text{sec})$.

Let us take a closer look at these rules:

- Rule 6 allows to any human party to generate a fresh term and to add it to his knowledge, along with the relevant tag t. This models, for example, the generation of a new password by a registering user.
- Rule 7 includes into the formalization the leakage of any information (both private and non) that makes up the human's knowledge database onto the network. A real-world example of this would be an involuntary password reveal by a user.

- Rule 8 allows a human to update his existing knowledge based on new information received from the network. For example, an unexperienced user may receive a phishing mail explaining that he must re-enter his credentials into a new website because of security reasons.

Note that all of the rules have action facts equivalent to the function symbols belonging to RoleActions. This will allow us to introduce rules that constrain the possible behaviours (thus effectively defining skilled humans) by including additional restrictions in the theory[2]. Technically, it would be possible to include any constraint to limit the possible action executed by a human agent, but the authors propose 4 very generic formulas that cover many social engineering threats:

`NoTell` for avoiding leakage of private information If we wanted to prevent a user H from revealing information labelled by *tag* to the rest of the network (for example a private key in an asymmetric cryptography context) during the execution of protocol P (intended as a set of protocol rewriting rules) we could include the following predicate (of course using the restriction construct):

$$\texttt{NoTell}(H, tag) := \forall \texttt{Send}(H, l, P, \langle t, m\rangle) \in tr \in traces(P), t', m' : \langle t, m\rangle \vdash_H \langle t', m'\rangle \Rightarrow t' \neq tag$$

This predicate ensures that H will never send the information tagged with t onto the network. Of course, this restriction can not be imposed on *all* private information of agent H: during authentication procedures, for example, a user may be legitimaetly prompted for the input of a password and this constraint would not allow it. For this reason, the following restriction has been introduced.

`NoTellExcept` for safe communication of credentials Let us assume that we have an action fact $\texttt{InitK}(H, \langle tag, x\rangle)$ that generalizes the `Start` action by denoting the knowledge of agent H of the term x with tag t at initialisation. We can exploit such fact to limit the human to only reveal *some* secret information to trusted parties over secure or authenticated channels:

$$\begin{aligned}\texttt{NoTellExcept}(H, tag, rtag) := \forall \texttt{Send}(H, l, P, \langle t, m\rangle) \in tr \in traces(P), m', R :\\ \texttt{InitK}(H, \langle rtag, R\rangle) \in tr \wedge \langle t, m\rangle \vdash_H \langle tag, m'\rangle \Rightarrow P = R \wedge (l = \text{sec} \vee l = \text{conf})\end{aligned}$$

This predicate ensures that all information labelled by *tag* is shared only with parties that we trust (since we include them in our initialisation knowledge). A practical implementation of this within a theory could require, for example, that any message labelled by the tag 'password' is only shared with an authentic platform (of which we know the URL saved under the tag *rtag*).

`NoGet` for protecting information integrity A user may be tricked by an attacker to update pre-existing (safety) information (for example through a realistic spear-phishing attack). It is not unrealistic, though, to assume that a reasonably cautious user may refrain to believe some new data regarding sensitive information received from the network. This situation can be modelled through the following predicate:

$$\texttt{NoGet}(H, tag) := \forall \texttt{Receive}(H, l, P, \langle t, m\rangle) \in tr \in traces(P), t', m' : \langle t, m\rangle \vdash_H \langle t', m'\rangle \Rightarrow t' \neq tag$$

Such constraint prevents any piace of knowledge tagged by *tag* to be updated from information retrieved from the network. A practical example of this would be to assume that an antivirus' customers only use the software built-in update mechanics and avoid accepting new "update links" received from internet.

[2] Note that here, technically, we are defining predicates. In practice, we implement them in Tamarin as restrictions, thus we will use these two terms interchangeably within the rest of this section

`ICompare` to enforce important security checks Some protocols like MTProto2 are relatively safe under the assumption of users checking QR codes for encryption to ensure the absence of Man-In-The-Middle attacks [MV23]. If we wanted to enforce similar checks within our system, we could include the following predicates:

$$\begin{aligned}\texttt{ICompare}(H, tag) := \forall \texttt{Receive}(H, l, P, \langle t, m\rangle) \in tr \in traces(P), m' :\\ \langle t, m\rangle \vdash_H \langle tag, m'\rangle \Rightarrow \texttt{InitK}(H, \langle tag, m'\rangle) \in tr\end{aligned}$$

Intuitively, this predicate requires that any piece of information received from the network and labelled with *tag* must be checked against the agent's initial knowledge.

Other predicates While the predicates proposed by the article cover a lot of real world scenarios, the list is far from comprehensive: for example, one may want to generalize the `NoTellExcept` restriction in order to allow an authenticated party to communicate to the user another trust platform with which he can share private information, or it may be useful to have define the `NoGet` predicate in order to avoid updating also knowledge created during the exchange and not only at initialisation. In any case, it is clear that extending a theory with such constraints is far from complex.

Proving properties The most convenient aspect of this approach is the fact that, by including the constraints on the behaviour of the untrained human through predicates, we can easily keep our usual formulas for the properties to check while quickly sweeping in or out the restrictions in order to verify different scenarios. Furthermore, by including the reference to the a particular human H within our predicates we are able to impose different degrees of freedom to various agents for a more general (and capillary) verification strategy. In particular, our approach distinguishes a particular human H for whom security properties are analyzed, while allowing an arbitrary number of untrained (or less skilled) humans in the network to verify whether errors of external parties may affect H's security guarantees.

4 Conclusions

In this article we firstly introduced the general problem of automated verification of cryptographic protocols in the Dolev Yao model, then we gave a general overview of the Tamarin Prover, mainly from a user's perspective. Although such outline covered most of the features provided by the tool, for a more comprehensive look at the subject we suggest consulting [Tea22]. Finally, we showed how to include various Social Engineering attacks within the considered threat model by formalizing human knowledge and agent errors through multisets rewriting rules and restrictions. This allows us to verify protocols that rely on multiple interactions with the users in a more realistic and complete way. Possible future works would include the development of additional predicates to model further assumptions and the ideation of a concrete system to implement the skilled humans approach.

References

[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.

[BRS15] David Basin, Sara Radomirovic, and Michael Schlapfer. A complete characterization of secure human-server communication. *2015 IEEE 28th Computer Security Foundations Symposium*, pages 199–213, 2015.

[BRS16] David Basin, Saa Radomirovic, and Lara Schmid. Modeling human errors in security protocols. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 325–340, 2016.

[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.

[DDKS17] Jannik Dreier, Charles Dumenil, Steve Kremer, and Ralf Sasse. Beyond subterm-convergent equational theories in automated verification of stateful protocols. In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust*, pages 117–140, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[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.

[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.

[Git23] Gitnux Gitnux. Social engineering 2023: Statistics and trends, Jul 2023.

[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.

[Low97] Gavin Lowe. A hierarchy of authentication specifications. In *Proceedings of the 10th IEEE Workshop on Computer Security Foundations*, CSFW '97, page 31, USA, 1997. IEEE Computer Society.

[Mei13] S. Meier. *Advancing automated security protocol verification.* PhD thesis, ETH, 2013.

[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.

[MV23] Marino Miculan and Nicola Vitacolonna. Automated verification of telegram's mtproto 2.0 in the symbolic model. *Computers and Security*, 126(C), 2023.

[Sch12] B. Schmidt. *Formal analysis of key exchange protocols and physical protocols.* PhD thesis, ETH, 2012.

[SK19] Fatima Salahdine and Naima Kaabouch. Social engineering attacks: A survey. *Future Internet*, 11(4), 2019.

[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.

[Tea22] The Tamarin Team. *Tamarin-Prover Manual: Security Protocol Analysis in the Symbolic Model*, 2022.

[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.