Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis

# Automatic Analysis of Communication Protocols with Human Errors

Andrina Denzler

Supervisors: Dr. Saša Radomirović, Dr. Ralf Sasse, and Lara Schmid
Professor: Prof. Dr. David Basin

October 7, 2016

## Abstract

The security of communication protocols is important in our everyday life. However, their formal verification does usually not consider the interaction with humans. The influence of untrained users is significant, as they are a potential source of error and cannot perform cryptographic operations. Human errors in security protocols can be modelled formally and analysed with TAMARIN, a symbolic security protocol verification tool. To thoroughly analyse protocols with human errors, a new protocol theory is manually specified and analysed for each considered set of errors. In this work, we automate this time-consuming and error-prone process. That is, we automatically generate all sets of errors and protocol theories, analyse them with TAMARIN, and collect the results. We apply our approach to three protocols in case studies and find multiple attacks with respect to different combinations of human errors.

**Acknowledgements**

First, I would like to thank my supervisors Saša Radomirović, Ralf Sasse, and Lara Schmid for their patient assistance and dedication to my work. They provided me with valuable input and advice in our regular meetings and I was able to benefit a lot from their knowledge. Further, I would like to thank Professor David Basin and the Information Security Group at ETH Zurich for the opportunity to write this thesis.

Last but not least, I would like to thank my family and friends for their support and encouragement during the last six months.

# Contents

# 1 Introduction

Humans interact with and participate in communication protocols on a daily basis. To do so securely, cryptographic protection is used. Unlike computers, most humans do not know the detailed protocol steps and cannot perform cryptographic operations on their own. Their lack of knowledge makes them prone to errors which an adversary may exploit in different attacks, for example, using social engineering. There is a model [3] for human errors in security protocols which allows them to be formally analysed with the automatic verification tool TAMARIN [1]. The desired properties of a protocol are proven or disproven under assumptions about the types of errors the human can make. Up until now, a thorough analysis of a protocol with different combinations of human errors required the manual specification of a new TAMARIN theory for every particular assumption about the human. In this thesis, we automate this time-consuming and repetitive process. For every security protocol and every set of human errors, we automatically find the boundary between secure and insecure with respect to different subsets of human errors.

**Contributions.** We introduce an algorithm to automate the analysis of security protocols with human errors. Each desired property of a communication protocol is analysed for various combinations of human errors. First, we assume that a human knows the protocol steps and that he does not make any errors. Then, we allow the human to make a fixed number of specific errors. Hence, we can examine which combinations of human errors are tolerable and which are not. Moreover, we implemented our algorithm in an automatic tool and use the TAMARIN prover as a back-end to analyse individual protocol theories. In our case studies, we analyse the protocols MP-Auth, Google 2-Step, and the Helbach code voting variant in the presence of human errors. We find multiple attacks on these protocols, i.e., we falsify certain security properties with respect to specific combinations of human errors.

**Outline.** In this thesis, we first present TAMARIN's formal protocol specifications and the human-error model in Section 2. In Section 3, we introduce our algorithm to generate and analyse different combinations of human errors with respect to a security property. In Section 4, we demonstrate our analysis tool in case studies. Finally, we conclude our work and discuss some ideas for future work in Section 5.

# 2 Preliminaries

Every communication protocol specifies two or more *roles* which determine the behaviour of computer and human agents. These agents exchange messages over different communication channels. An adversary in the Dolev-Yao model can block the entire communication, and read, modify, and send individual messages. In TAMARIN, the communication network is per default controlled by a Dolev-Yao adversary.

In this section, we explain the preliminaries needed to understand automatic protocol analysis with human errors. First, we introduce the notation we use to model protocols. Then, we present TAMARIN's security protocol model [6] and its input language [2]. Finally, we explain our concept of human errors.

## 2.1 Alice&Bob Notation

We represent communication protocols in the extended Alice&Bob notation introduced by *Schläpfer* [5]. Our notation supports symbols for the following types of communication channels:

- To express that a message $m$ is sent from $A$ to $B$ over an insecure channel, we write $A \circ\!\!\rightarrow\!\!\circ B : m$. An adversary can both send and read messages on insecure channels.
- An adversary cannot modify messages communicated over authentic channels, denoted by $\bullet\!\!\rightarrow\!\!\circ$. However, he is able to read these messages.
- $\circ\!\!\rightarrow\!\!\bullet$ denotes a confidential channel. That is, the adversary cannot read the content sent over these channels, but can send its own messages.
- Secure, i.e., authentic and confidential, channels are denoted by $\bullet\!\!\rightarrow\!\!\bullet$. The adversary can neither send nor read messages on secure channels, but he is able to block the communication.
- Human-interaction channels $\mathrm{h}\!\!\rightarrow\!\!\mathrm{h}$ differ from the other channel types, as they represent the direct interaction between a human user and his computer. In particular, we use $A \ \mathrm{h}\!\!\rightarrow\!\!\mathrm{h} \ B : m$ to express that a message $m$ is directly delivered from $A$ to $B$ and that either $A$ or $B$ describes a human role. Thus, the adversary cannot learn $m$ nor is he able to modify or replay the message. In contrast to secure channels, he cannot block human-interaction channels.

We write $A$ : knows($m$) to express the initial knowledge $m$ of $A$. We use $A \circ\!\!\rightarrow\!\!\circ B$ : fresh($m$).$m$ to denote that $A$ generates a fresh name $m$ and then sends $m$ to $B$. To express that $A$ compares the two terms $m$ and $m'$, we write $A$ : compares($m, m'$). These comparisons have to be added explicitly to the protocol in human roles. We use $A \circ\!\!\rightarrow\!\!\circ B : m \ / \ m'$ to indicate that the message sent and the message received are interpreted differently. The term $m$ determines the sender's view, while the term $m'$ defines how the received message is parsed.

**Example 2.1.** Consider the protocol specification of a simple protocol given in Figure 2.1. The protocol specifies the roles of two agents. The term knows($S, c$) in the first line indicates that an agent in role $H$ initially knows $S$ and the code $c$. The second line denotes that $S$ initially only knows $H$. The initiator $H$ sends $c$ from his knowledge over an insecure channel to $S$. Then, $S$ generates a fresh message $m$ and sends $m$ over a confidential channel back to $H$.

$$\begin{array}{lrl}
0. & \mathrm{H}: & \mathrm{knows}(S, c) \\
0. & \mathrm{S}: & \mathrm{knows}(H) \\
1. & \mathrm{H} \circ\!\!\rightarrow\!\!\circ \mathrm{S}: & c \\
2. & \mathrm{S} \circ\!\!\rightarrow\!\!\bullet \mathrm{H}: & \mathrm{fresh}(m).m
\end{array}$$

Figure 2.1: Example protocol in extended Alice&Bob notation.

## 2.2 Tamarin

We use TAMARIN [1] to automatically analyse communication protocols. The TAMARIN prover is a verification tool that symbolically analyses protocols. It supports both falsification and unbounded verification. TAMARIN's input is the model of a security protocol together with the protocol's desired properties. A protocol is modelled as a multiset rewriting system and every property can be verified or falsified.

### 2.2.1 Security Protocol Model

Cryptographic messages are modelled using an order-sorted term algebra. There is a sort *msg* and two incomparable subsorts *fresh* and *pub* for fresh and public names. To model cryptographic operators, TAMARIN supports a user-defined signature which describes a set of function symbols. *Terms* are built over the signature, all fresh and public names, and the set of variables.

A *fact* is of the form $\mathsf{F}(t_1, \ldots, t_k)$ for a $k$-ary fact symbol $\mathsf{F}$ and terms $t_i$. The set of facts is partitioned into linear and persistent facts. They model exhaustible and inexhaustible resources, respectively. There are built-in facts In, Out, and Fr to model the communication over the untrusted network and to model freshly generated values.

**Definition 2.2.** [6] A linear fact In$(m)$ denotes that the adversary has sent the message $m$, which can be received by an agent participating in the protocol.

**Definition 2.3.** [6] A linear fact Out$(m)$ denotes that the protocol has sent the message $m$ to the network, which can be received by the adversary.

**Definition 2.4.** [6] A linear fact Fr$(n)$ denotes that the fresh name $n$ was freshly generated.

A *multiset rewriting rule* is of the form $l \,{-}[\, a \,]{\rightarrow}\, r$, where $l$, $a$, and $r$ are multisets of facts and denote the rule's premise, action, and conclusion, respectively. The action labels the rule and is a set of so called action facts.

There is a built-in rule which produces fresh names and can be applied in any system state. Furthermore, there are built-in message deduction rules which allow the Dolev-Yao adversary to receive and send messages, to learn public names, to generate fresh names of its own, and to apply functions from the signature to known messages. Additionally, rules are used to model the protocol and the adversary's capabilities.

A *protocol* is a finite set of rewriting rules. For each role in the protocol, we need one rewriting rule for every protocol step the role is involved in. In particular, we can directly translate a protocol in Alice&Bob notation to multiset rewriting rules. We extend TAMARIN's security protocol model in Section 2.3 and explain there how to translate the example protocol shown in Figure 2.1 to TAMARIN input.

The execution of a protocol is modelled as a labeled transition system. A system state is a finite multiset of facts. The execution starts in the initial system state which is the empty multiset. Each transition between two subsequent system states is defined by a rewriting rule. A rule's instance can be applied to the current state $S$ if its premise occurs in $S$. Then, linear facts in the rule's premise are removed from $S$ and all instantiated facts in the rule's conclusion are added to $S$. That is, linear facts can only be consumed once. Persistent facts, on the other hand, can be consumed arbitrarily often, as they are never removed from the state.

A *trace* is a sequence of actions that occur during a protocol's execution. Thereby, a trace is a sequence of multisets of facts. *Security properties* are specified with first-order logic formulas over traces. Formulas can be quantified over both messages and timepoints. For this reason, there is an additional sort *temp* for timepoints.

### 2.2.2 Proof Search

TAMARIN has both an *automated* and an *interactive* mode to construct proofs. In this work, we use the automated proof search. When TAMARIN analyses the desired trace property of a protocol, there are three possible outcomes:

**Verification.** In case Tamarin's automated proof search terminates and the stated trace property holds for the given protocol, it returns a proof of correctness.

**Falsification.** If the proof search terminates, but the property does not always hold for the protocol, Tamarin returns a trace representing an attack, i.e., a counterexample to the property.

**Timeout.** Sometimes Tamarin cannot find a proof at all or at least not within a given amount of time. Properties of protocols are an undecidable problem in general.

Tamarin supports different heuristics to influence the proof search. The automated mode orders unsolved tasks by priorities which determine how the proof search should proceed. The heuristics help to sort these unsolved tasks in different ways. One of these heuristics is the *oracle* heuristic which allows the user to entirely customise prioritisation for one protocol.

### 2.2.3 Input Language

Tamarin processes its own security protocol theory format. The complete formal syntax of this format is explained in the Tamarin manual [2]. Each theory is stored in a `.spthy` file and has the following form:

```
theory TheoryName
begin
   /* body */
end
```

The header of a theory starts with the keyword `theory` followed by the theory's name, here `TheoryName`, and the keyword `begin`. The theory must be delimited with `end`. The file can be annotated with C-style comments. That is, line comments start with `//`, and multi-line comments are delimited by `/*` and `*/`. Next, we explain different parts declared in a security protocol theory.

**Cryptographic Messages.** A cryptographic message is a constant `c`, a variable `x` or a message `f(m1,...,mn)` for the `n`-ary function symbol `f` and messages `m1` to `mn`. In Tamarin, public constants are denoted as string constants `'c'`. Recall that there is a sort `msg` for variables, two incomparable subsorts `fresh` and `pub`, and a disjoint sort `temp` for timepoint variables. Tamarin uses prefixes to denote the sort of variables:

- `~x` denotes a variable of sort `fresh`.
- `$x` denotes a variable of sort `pub`.
- `#i` denotes a variable of sort `temp`.
- `m` denotes a variable of sort `msg`.

Function symbols `f1`, ..., `fn` with arity `a1`, ..., `an`, respectively, are declared using the following syntax:

```
functions:  f1/a1, ..., fn/an
```

Equational theories allow to model relations between functions and their input values. In TAMARIN, equations are added to a protocol theory in the following form:

```
equations:  lhs1 = rhs1, ..., lhsn = rhsn
```

Both the left-hand-side `lhs` and the right-hand-side `rhs` are terms such that the equations are subterm-convergent. That is, the right-hand-side either does not contain variables or is a proper subterm of the left-hand-side.

TAMARIN provides built-in function signatures and the corresponding equational theories for some common functions such as hashing, encryption, and signing.

**Example 2.5.** The theory below models a public key encryption scheme. The binary function symbols `adec` and `aenc` model asymmetric decryption and encryption, respectively. The unary function symbol `pk` models the computation of a public key from a private key. We define the semantic dependencies between the functions with the following equation:

```
functions:  adec/2, aenc/2, pk/1
equations:  adec(aenc(m,pk(k)),k) = m
```

The above theory can also be enabled using TAMARIN's built-in functions for asymmetric-encryption with the following line.

```
builtins:  asymmetric-encryption
```

**Model Specification.** In the previous section, we formally introduced facts and rewriting rules. Here, we explain how they are written in TAMARIN's input language.

**Definition 2.6.** A *fact* is of the form `F(t1,...,tn)` for the `n`-ary fact symbol `F` and terms `t1` to `tn`.

Facts have a fixed arity and start with an upper-case letter. They are not declared explicitly, but need to be used consistently. Persistent facts are prefixed with an exclamation mark, while linear facts are not.

**Definition 2.7.** A *rule* starts with the keyword `rule` and specifies a multiset rewriting rule. Every rule has a unique name. Therefore, rules are of the form:

```
rule ident:
   [ premise ] --[ action ]-> [ conclusion ]
```

When the rule has no action, we can replace the arrow notation `--[]->` with its short form `-->`.

**Property Specification.** Protocol properties are specified as security properties over traces. Properties whose satisfiability or validity we wish to check are expressed as lemmas.

**Definition 2.8.** A *lemma* starts with the keyword `lemma` and specifies a first-order formula. Every lemma has a name, and can be annotated with attributes and a trace quantifier.

Next, we give an example of a lemma. However, we do not explain the syntax any further and refer to the Tamarin manual [2].

**Example 2.9.** The lemma `Entity_authentication` states that whenever agent `S` commits to agent `'Human'` at timepoint `k`, `'Human'` has taken some action between the start of `S` and the claim. The lemma expresses the entity authentication property defined in Definition 4.1.

```
lemma Entity_authentication:
   "All S m #k. Commit(S,'Human',m) @k ==>
   (Ex #i #j. StartS(S) @i & H('Human') @j & i<j & j<k)"
```

Tamarin supports two different *trace quantifiers*: The `all-traces` and the `exists-trace` quantifier. A trace quantifier can be added to the lemma in front of its formula. For a lemma named `LemmaName` with the trace quantifier `Quantifier`, the lemma specification starts with `lemma LemmaName: Quantifier`. By default, Tamarin analyses whether the property holds for all traces. When a lemma is annotated with the `exists-trace` quantifier, Tamarin checks whether there exists a trace on which the property holds. Such existence proofs are useful sanity checks to guarantee the functionality of protocol models.

There is the possibility to change a lemma's meaning by annotating it with *attributes*. In Tamarin, they are added to the lemma in square brackets after its name. For a lemma named `LemmaName` with attributes

`Annotation1` and `Annotation2`, the lemma specification starts with `lemma LemmaName [Annotation1,Annotation2]`. A full overview of all attributes can be found in the Tamarin manual [2]. For the purpose of this thesis, we introduce the `reuse` and the `typing` attribute:

- A `reuse` lemma is used in the proofs of all lemmas that syntactically follow it. That is, Tamarin assumes the `reuse` lemma holds and may use it to complete the proof of any lemma syntactically following, even if the `reuse` lemma itself is not proven.
- All `typing` lemmas are used during Tamarin's precomputation and thus, are utilised in the proofs of all non-`typing` lemmas.

We denote both `reuse` and `typing` lemmas as *helper lemmas.* They are special in the sense that the proofs of non-helper lemmas are only valid if all helper lemmas are verified.

Properties who restrict the set of traces considered in the analysis are expressed as axioms. They are used to model assumptions on the protocol's execution, i.e., specific traces can be excluded from the analysis.

**Definition 2.10.** An *axiom* starts with the keyword `axiom` and specifies a first-order formula. Every axiom has a name.

**Example 2.11.** The axiom `One_failure` forces the `Failure()` action to be unique, i.e., there is at most one `Failure()` action in every trace. The axiom will be explained in more detail in Section 2.3.

```
axiom One_failure:
   "All #i #j. Failure() @i & Failure() @j ==> #i = #j"
```

The syntax of these formulas is the same for both lemmas and axioms. Again, we refer to the Tamarin manual [2] for more details.

## 2.3 Human Errors

We use a model for human errors [3] that allows us to analyse communication protocols with human errors and even provides tool support using Tamarin. Recall that a role specification describes the behaviour of an agent in a protocol.

**Definition 2.12.** [3] A *human error* in a protocol execution is any deviation of the human from his or her role specification. Such a human is said to be *fallible.* A human that does not deviate from his role specification is said to be *infallible.*

A fallible human allows more system behaviours than an infallible human by deviating from the protocol specification. These additional system behaviours may lead to an attack on the protocol.

We informally explain two approaches to model fallible humans: The skilled human approach and the rule-based human approach. In this thesis, we only consider the formal model of the skilled human [5].

**Skilled Humans.** This approach starts from an infallible human agent. We know that the infallible human precisely follows the role specification (Definition 2.12). This models a human user who knows the protocol. The skilled human also follows the protocol's steps. However, a skilled human can make a fixed number of errors. The skilled human allows more system behaviours which makes him weaker than an infallible human.

**Rule-Based Humans.** In this approach, we start from an untrained human agent. The untrained human has no information on the role specification and can perform arbitrary actions during the protocol's execution. The rule-based human has no knowledge about the role specification either. Nevertheless, he follows a set of guidelines. These guidelines are rules that restrict the rule-based human's behaviours and consequently the system behaviours.

To model skilled humans, we present a model of human capabilities [3], [5]: Humans may send and receive messages over channels, concatenate and split terms, compare two terms, and generate random terms. Humans cannot perform cryptographic operations, i.e., they cannot encrypt and decrypt messages without the help of a computer. The number of terms a human is able to remember is not limited.

In a next step, we identify potential failure modes [5]. These failures are based on all the tasks a human agent can perform in his role specification. A skilled human agent may fail to perform the following tasks:

- Send or receive terms.
- Concatenate or split terms.
- Compare terms.
- Generate fresh terms.
- Remember terms.
- Execute a protocol in the expected order.

The set of all *failure identifiers* is derived from the identified failure modes. It is denoted by $F_{\mathrm{ID}}$ and an overview is given in Table 2.1. Next, we explain what goes wrong in each of these errors.

| $f_{\mathrm{ID}} \in F_{\mathrm{ID}}$ | Description |
|---|---|
| *msc* | Message sending confusion. |
| *csc* | Channel sending confusion. |
| *mrc* | Message receiving confusion. |
| *nlc* | No learning confusion. |
| *crc* | Channel receiving confusion. |
| *icc* | Ignoring comparison confusion. |
| *wrc* | Weak randomness confusion. |
| *soc* | Step order confusion. |
| *src* | Step repetition confusion. |

Table 2.1: Failure identifiers in $F_{\mathrm{ID}}$ [5].

**Message sending confusion** (*msc*) models a human who sends an arbitrary term instead of the specified one.

**Channel sending confusion** (*csc*) models a human who uses the wrong channel to send a term if more than one channel is available.

**Message receiving confusion** (*mrc*) models a human who learns a different term than the specified one. The new term can be public or freshly generated.

**No learning confusion** (*nlc*) models a human who is expected to receive a specific term, but does not learn a term at all.

**Channel receiving confusion** (*crc*) models a human who confuses incoming channels if multiple channels exist.

**Ignoring comparison confusion** (*icc*) models a human who ignores to perform a comparison properly and further executes the protocol.

**Weak randomness confusion** (*wrc*) models a human who generates a weak random term when he is supposed to provide a fresh random value.

**Step order confusion** (*soc*) models a human who confuses the expected order of protocol steps. This, for example, allows the user to skip one or more protocol steps.

**Step repetition confusion** (*src*) models a human who executes a protocol step repeatedly.

### 2.3.1 Extended Security Protocol Model

We extend the TAMARIN model with user-defined facts AgentState, Snd, Rcv, and Fresh to model agents in a protocol [5].

**Definition 2.13.** [5] An AgentState fact is of the form $\mathsf{AgentState}(A, c, n)$, where $A$ is an agent's name, $c$ refers to the role step the agent is in, and $n$ is the agent's knowledge at that step.

To model how protocol agents access channels, we use Schläpfer's set of channel rules [5], and the two facts Snd and Rcv. I, A, C, S, and HI indicate the channel property for an insecure, authentic, confidential, secure, and human-interaction channel, respectively.

**Definition 2.14.** A Snd fact is of the form $\mathsf{Snd}(p, A, B, m)$ where $A$ and $B$ are the names of two agents, $p \in \{\mathsf{I}, \mathsf{A}, \mathsf{C}, \mathsf{S}, \mathsf{HI}\}$, and $m$ is the message that agent $A$ sends to receiver $B$ over a channel with property $p$.

**Definition 2.15.** A Rcv fact is of the form $\mathsf{Rcv}(p, A, B, m)$ where $A$ and $B$ are the names of two agents, $p \in \{\mathsf{I}, \mathsf{A}, \mathsf{C}, \mathsf{S}, \mathsf{HI}\}$, and $m$ is the message that agent $B$ receives from sender $A$ over a channel with property $p$.

Every message $m$ in a Snd or Rcv fact is of the form $\langle n, t\rangle$ where $n$ is a message tag, i.e., a constant identifier, and $t$ is the actual message.

**Definition 2.16.** A Fresh fact is of the form $\mathsf{Fresh}(A, n)$ where $A$ is an agent's name and $n$ is the term that $A$ freshly generated.

The rule FR allows agents to generate fresh constants. To later cover weak randomness, we explicitly state which agent generates which fresh term.

```
/* Agent fresh rule. */
rule FR:
   [ Fr(~x) ] --[  ]-> [ Fresh($A,~x) ]
```

We handle human knowledge with !HK facts. They represent all terms a human agent may create, i.e., the initial knowledge, all terms learnt during a protocol execution, and all derivable terms a human can produce in a protocol execution.

**Definition 2.17.** [5] A !HK fact is of the form $!\mathsf{HK}(A, t)$ where $A$ is a human agent's name and $t$ is a term known to $A$.

The facts Compare, Verified, and Eq model the explicit comparisons performed by human agents.

**Definition 2.18.** A Compare fact is of the form $\mathsf{Compare}(A, m, m')$ where $A$ is a human agent's name, and $m$ and $m'$ are the two terms that agent $A$ has to compare.

**Definition 2.19.** A Verified fact is of the form $\mathsf{Verified}(A, m, m')$ where $A$ is a human agent's name, and $m$ and $m'$ are the two terms that $A$ successfully compared.

**Definition 2.20.** An Eq action fact is of the form $\mathsf{Eq}(m, m')$ where $m$ and $m'$ are two terms. Eq facts are used to mark equal terms in a trace.

The rule CP models an agent A who compares the two terms m1 and m2. The rule is labeled with an Eq fact.

```
/* Compare rule. */
rule CP:
   [ Compare($A,m1,m2) ]
   --[ Eq(m1,m2) ]->
   [ Verified($A,m1,m2) ]
```

To guarantee that all comparisons are performed correctly, we restrict the set of traces that we examine with the following axiom:

```
axiom Comparison:
   "All m1 m2 #i. Eq(m1,m2) @i ==> m1 = m2"
```

With the axiom Comparison, we restrict the set of traces to those where the two terms m1 and m2 are actually equal whenever they are marked equal.

Recall that protocols are specified by a finite number of rewriting rules. There is a unique setup rule that initialises the protocol roles and creates the agents' initial knowledge. Each role is characterised by a set of protocol rules.

**Definition 2.21.** A *setup rule* $l \, -[\, a \,]\!\!\rightarrow r$ is a rewriting rule where:

- Only Fresh and Fr facts occur in $l$.
- For every role in the protocol, there is an AgentState fact in $r$.

**Definition 2.22.** A *protocol rule* $l \, -[\, a \,]\!\!\rightarrow r$ is a rewriting rule where:

- Only Rcv, Fresh, Verified, and AgentState facts occur in $l$.
- Only Snd, Compare, !HK, and AgentState facts occur in $r$.
- One AgentState fact occurs in $l$ and at most one AgentState fact occurs in $r$ for the same agent.

Recall that !HK facts are used to represent human knowledge. The initial human knowledge and all terms learnt during a protocol execution have to be explicitly stated in the protocol specification. More precisely, for every term $t$ in the initial knowledge of human $A$, we add a $!\mathsf{HK}(A, t)$ fact to the setup rule's conclusion. For every term $t$ that human $A$ freshly generates or receives in a protocol rule, we add a $!\mathsf{HK}(A, t)$ fact to the rule's conclusion. Now, we show how a protocol in Alice&Bob notation can be modelled in Tamarin's input language.

**Example 2.23.** Consider the protocol in extended Alice&Bob notation shown in Figure 2.1 on page 4. The protocol in TAMARIN's input language is given in Figure 2.2. The setup rule can be applied in any system state and sets up the initial knowledge of both human `H` and server `S`. The first `AgentState` fact denotes state `'H_0'` of agent `H` with the initial knowledge `<S,code>`. The second `AgentState` fact denotes state `'S_0'` of agent `S` with the initial knowledge `H`. The two `HK` facts explicitly state the initial knowledge of human `H`. The premise of the protocol rule `H_0` is that `H` is in state `'H_0'` with his knowledge `<S,code>`. The first fact in the conclusion expresses that agent `H` proceeds to the next state with the same knowledge. The second fact in the conclusion depicts that `H` sends `code` with the message tag `'M_1'` over an insecure channel to `S`. The protocol rule `S_1` describes that agent `S` is in state `'S_0'`, receives a message over an insecure channel, and generates a fresh term `m`. Furthermore, `S` sends message `'M_2'` over a confidential channel back to `H`, updates his knowledge with the learned terms, and proceeds to the next agent state. The premise of rule `H_2` consists of the current agent state, and a `Rcv` fact which expresses that `H` receives a message over a confidential channel. In the rule's conclusion, the knowledge in the agent state of `H` is updated, and the human knowledge is explicitly updated with the `HK` fact.

To model erroneous human behaviour, the protocol contains further multiset rewriting rules. We use a special failure rule and a set of human-error rules to select and include human errors, respectively. To do so, we need the user-defined facts Failed, Failure, and Fail.

**Definition 2.24.** A Failed action fact is of the form $\mathsf{Failed}(A)$ where $A$ is a human agent's name. Failed facts are used to mark human-error rules in a trace.

**Definition 2.25.** A Failure action fact is of the form $\mathsf{Failure}()$. Failure facts are used to mark failure rules in a trace.

**Definition 2.26.** [5] A Fail fact is of the form $\mathsf{Fail}(A, f_{\mathrm{ID}})$ where $A$ is a human agent's name and $f_{\mathrm{ID}} \in F_{\mathrm{ID}}$ is a failure identifier. Fail facts model the assumption that a human user may fail in a specific way.

Human-error rules model a human agent's ability to deviate from his role specification. We denote the set of all human-error rules by $\mathcal{E}_{all}$.

**Definition 2.27.** A *human-error rule* $l \,{-}[\, a \,]{\rightarrow}\, r$ is a rewriting rule where:

- One $\mathsf{Fail}(A, \_)$ fact occurs in $l$.
- One $\mathsf{Failed}(A)$ action occurs in $a$.

Each failure identifier in $F_{\mathrm{ID}}$ represents a set of human-error rules. Next, we explain the human-error rules identified by the failure identifiers *msc*, *icc*, and *soc* in detail.

```
/* Setup rule. */
rule Setup:
[  ]
-->
[ AgentState($H,'H_0',<$S,$code>),
  AgentState($S,'S_0',<$H>),
  !HK($H,$S), !HK($H,$code) ]

/* Protocol specification rules. */
rule H_0:
[ AgentState($H,'H_0',<$S,$code>) ]
-->
[ AgentState($H,'H_1',<$S,$code>),
  Snd('I',$H,$S,<'M_1',$code>) ]

rule S_1:
[ AgentState($S,'S_0',<$H>),
  Rcv('I',$H,$S,<'M_1',code>), Fresh($S,~m) ]
-->
[ AgentState($S,'S_2',<$H,code,~m>),
  Snd('C',$S,$H,<'M_2',~m>) ]

rule H_2:
[ AgentState($H,'H_1',<$S,$code>),
  Rcv('C',$S,$H,<'M_2',m>) ]
-->
[ AgentState($H,'H_3',<$S,$code,m>), !HK($H,m) ]
```

Figure 2.2: Example protocol in TAMARIN's input language.

The following human-error rule models message sending confusion and is identified by the failure identifier *msc*. Human agent `A` sends the term `m2` from his knowledge instead of the specified term `m1` to an agent `B` using a channel with property `p`.

```
/* Message sending confusion. */
rule E_msc:
   [ Fail($A,'msc'), !HK($A,m2), Snd(p,$A,$B,<cid,m1>) ]
   --[ Failed($A) ]->
   [ Snd(p,$A,$B,<cid,m2>) ]
```

The failure identifier *icc* represents ignoring comparison confusion. It is modelled with the next rewriting rule where human agent `A` omits the comparison of the two terms `m1` and `m2` when he is supposed to compare them, denoted by `Compare($A,m1,m2)`.

```
/* Ignoring comparison confusion. */
rule E_icc:
   [ Fail($A,'icc'), Compare($A,m1,m2) ]
   --[ Failed($A) ]->
   [ Verified($A,m1,m2) ]
```

The human-error rule for step order confusion (*soc*) allows a user to skip one or several protocol steps. However, this is only possible if the computer agents support this or if the protocol step is non-blocking, such as a verification step. Whenever human agent `A` is in a state with identifier `c1` and knowledge `n`, he may switch to another state identified with `c2` and the same knowledge `n`.

```
/* Step order confusion. */
rule E_soc:
   [ Fail($A,'soc'), AgentState($A,c1,n), !HK($A,c2) ]
   --[ Failed($A) ]->
   [ AgentState($A,c2,n) ]
```

Every time a human-error rule is applied in a protocol run, the Fail fact in its premise is consumed. For this purpose, we add a failure rule to the protocol which can generate Fail facts.

**Definition 2.28.** [5] A *failure rule* $l -[\, a \,]\!\rightarrow r$ is a rewriting rule where:

- $l$ is empty.
- One Failure() action occurs in $a$.
- Only Fail facts occur in $r$.

A protocol's specification must include a specific set of Fail facts in the failure rule, to analyse the protocol with respect to this set of human errors. To control the number of generated Fail facts, the failure rule may only be executed once. We restrict the set of traces that we examine with the following axiom that we already saw in Example 2.11.

```
axiom One_failure:
   "All #i #j. Failure() @i & Failure() @j ==> #i = #j"
```

With the axiom `One_failure`, we restrict the set of traces to those who contain at most one failure rule, as the `Failure()` action is unique. Note that the failure rule may add many Fail facts.

# 3 Automatic Analysis of Communication Protocols with Human Errors

Our tool supports a systematic analysis process. The tool's input is the formal specification of a security protocol and a set human errors. We test the protocol automatically against various combinations of the given human errors. These combinations are limited by a lower and an upper bound on the number of failures. The tool can be divided into different components:

- During the precomputation, we parse the provided input, prepare the input for further usage, and store the information in an intermediate representation.
- We generate a protocol specification file based on the intermediate representation for every specific set of errors. In particular, we add the human-error rules to the original protocol specification and include the corresponding set of Fail facts in the failure rule.
- Our tool uses TAMARIN as a back-end to analyse every protocol theory file and terminates TAMARIN's execution if the automated proof search cannot find a proof within the given amount of time.
- The search algorithm determines the order in which the protocol is tested against different combinations of human errors.
- We collect the obtained results, let them influence the further search strategy, and summarise them in the end.

The tool's source code is available at [4]. Below, we explain the tool's input and output, and present how the search algorithm works in detail.

## 3.1 Input Requirements

The tool expects a security protocol theory and a set of human errors in the skilled human error class. Additionally, we allow the user to influence the computation by providing optional arguments. Running our tool without arguments will show a help message with an overview on how to use these arguments.

### 3.1.1 Security Protocol Theory

The formal specification of a security protocol has to be specified in a TAMARIN file. We recommend to run TAMARIN on the input file in isolation before analysing it with our tool to ensure it fulfils its expected properties without human errors.

There are some syntactic requirements on the protocol specification which do not restrict the model of a skilled human:

- We assume there are no Failure and Fail facts present in the protocol specification and thus, also no failure and human-error rules. The failure rule is added to the protocol specification by our tool, and the human-error rules are specified by the user in a separate file.
- Several human-error rules rely on a correctly updated human knowledge. Otherwise it is not possible to thoroughly analyse these failures. The user has to make sure that !HK facts are present in the protocol specification wherever the human agent's knowledge has to be updated. Recall that the initial human knowledge and all terms learnt during a protocol execution have to be stated explicitly, as we explained in Section 2.3.
- Even though it is not necessary, we recommend to use mutually distinct lemma names, i.e., no lemma's name is a prefix of another lemma's name. Our implementation keeps all lemmas in the theory's body and proves lemmas selectively with TAMARIN's `--prove=LEMMAPREFIX` flag. The problem with not having mutually distinct names is that TAMARIN is slower, and in the worst case, might even fail to complete a proof, as it begins verification of all lemmas with that prefix.

### 3.1.2 Set of Human Errors

The set of human errors has to be specified in a separate *failure file* for which we introduce the `.f` filename extension. A failure file contains human-error rules in valid TAMARIN syntax and can be annotated with C-style comments. Recall that line comments start with `//`, and everything between `/*` and `*/` is a multi-line comment.

**Example 3.1.** Figure 3.1 depicts a valid failure file with two human-error rules. The failure identifiers *msc* and *icc* were explained in Section 2.3.

Users who run our tool have to make sure that the human-error rules in the failure file comply with the remaining protocol specification in the TAMARIN file, i.e., facts need to be used consistently.

```
/* Message sending confusion. */
rule E_msc:
[ Fail($A,'msc'), !HK($A,m2), Snd(p,$A,$B,<cid,m1>) ]
--[ Failed($A) ]->
[ Snd(p,$A,$B,<cid,m2>) ]

/* Ignoring comparison confusion. */
rule E_icc:
[ Fail($A,'icc'), Compare($A,m1,m2) ]
--[ Failed($A) ]->
[ Verified($A,m1,m2) ]
```

Figure 3.1: Example failure file with two human-error rules.

### 3.1.3 Optional Arguments

Our tool allows the following optional arguments to refine an analyser run:

- A minimum and maximum number of Fail facts: We restrict the number of failures by a lower and an upper bound. Thereby, the search space can be adjusted. The minimum has to be a non-negative integer. Naturally, the maximum should be larger than the minimum.
- A timeout for a single run of TAMARIN: We specify how long our tool waits for a result from TAMARIN before it terminates the background process and proceeds with its computation.
- An oracle flag: In case this flag is set, all calls to TAMARIN use the oracle heuristic. This requires an oracle file in the working directory.

## 3.2 Algorithm

We present an algorithm to generate and analyse different combinations of human errors in a controlled manner. This allows us to consider the result of previously analysed combinations when preparing the next combination. The algorithm operates on all lemmas one at a time, i.e., lemmas are analysed separately in order of their appearance in the input file. The lemma's attributes and trace quantifier also influence the search order. To precisely arrange different combinations of human errors, we introduce task identifiers and explain how different combinations correlate.

**Definition 3.2.** A *task identifier* is a multiset of failure identifiers, i.e., a set of human errors and their multiplicities. The empty multiset represents an infallible human.

We use the superscript $\sharp$ to denote operations on multisets. For example, we use $\subseteq^\sharp$ for the subset relation on multisets. For a given set of failure identifiers $F_{\mathrm{ID}}$, we denote the set of all task identifiers by $T_{\mathrm{ID}}$. This set is partially ordered by the subset relation on multisets. The partially ordered set is denoted by $(T_{\mathrm{ID}}, \subseteq^\sharp)$.

**Example 3.3.** Given the set $F_{\mathrm{ID}} = \{a, b\}$ of failure identifiers. Figure 3.2 shows the Hasse diagram representing the partially ordered set $(T_{\mathrm{ID}}, \subseteq^\sharp)$ over $F_{\mathrm{ID}}$.

$$\begin{tikzcd}[column sep=tiny]
\ddots & & \iddots & & \ddots & & \iddots & & \ddots & & \iddots & & \ddots & & \iddots \\
& \{a,a,a\} & & & & \{a,a,b\} & & & & \{a,b,b\} & & & & \{b,b,b\} & \\
& & & \{a,a\} \arrow[ull, dash] \arrow[urr, dash] & & & & \{a,b\} \arrow[ull, dash] \arrow[urr, dash] & & & & \{b,b\} \arrow[ull, dash] \arrow[urr, dash] & & & \\
& & & & & \{a\} \arrow[ull, dash] \arrow[urr, dash] & & & & \{b\} \arrow[ull, dash] \arrow[urr, dash] & & & & & \\
& & & & & & & \emptyset \arrow[ull, dash] \arrow[urr, dash] & & & & & & &
\end{tikzcd}$$

Figure 3.2: The partially ordered set $(T_{\mathrm{ID}}, \subseteq^\sharp)$ over $F_{\mathrm{ID}} = \{a, b\}$.

**Definition 3.4.** $X$ is a *direct submultiset* of $Z$ if $X \subsetneq^\sharp Z$ and there is no multiset $Y$ such that $X \subsetneq^\sharp Y$ and $Y \subsetneq^\sharp Z$.

**Definition 3.5.** $X$ is a *direct supermultiset* of $Z$ if $X \supsetneq^\sharp Z$ and there is no multiset $Y$ such that $X \supsetneq^\sharp Y$ and $Y \supsetneq^\sharp Z$.

For example, let $X := \{a, b\}$ and $Y := \{a, a, b\}$. $X$ is a direct submultiset of $Y$ and vice versa $Y$ is a direct supermultiset of $X$. The direct sub- and supermultisets of a specific multiset can be computed with the multiset sum and multiset difference operators we explain on an example below.

**Example 3.6.** The *multiset sum* $\uplus^\sharp$ of the two multisets $\{a, a, b\}$ and $\{a\}$ is $\{a, a, b\} \uplus^\sharp \{a\} = \{a, a, a, b\}$.

**Example 3.7.** The *multiset difference* $\setminus^\sharp$ of $\{a, a, b\}$ and $\{a\}$ is the multiset $\{a, a, b\} \setminus^\sharp \{a\} = \{a, b\}$.

Assume the failure identifiers $F_{\mathrm{ID}} = \{f_1, .., f_n\}$ and a task identifier $t_{\mathrm{ID}}$ over $F_{\mathrm{ID}}$. We denote the set corresponding to $t_{\mathrm{ID}}$ by $\overline{t_{\mathrm{ID}}}$, i.e., $\overline{t_{\mathrm{ID}}}$ is the set of all distinct elements in $t_{\mathrm{ID}}$. The task identifier's direct submultisets are $t_{\mathrm{ID}} \setminus^\sharp t$ for all $t \in \overline{t_{\mathrm{ID}}}$. The direct supermultisets of $t_{\mathrm{ID}}$ are given by $t_{\mathrm{ID}} \uplus^\sharp f_i$ for all $i \in \{1, .., n\}$. Note that each two multisets $X$ and $Y$ such that $X$ is a direct submultiset of $Y$ are connected in the Hasse diagram.

**Example 3.8.** Recall the partially ordered set $(T_{\mathrm{ID}}, \subseteq)$ over $F_{\mathrm{ID}} = \{a, b\}$ given in Figure 3.2. Now, Figure 3.3 shows the extract of this Hasse diagram that depicts both the direct submultisets and the direct supermultiset of the task identifier $\{a, a, b\}$ over $F_{\mathrm{ID}}$.

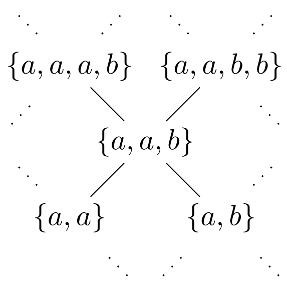

Figure 3.3: The task identifier $\{a, a, b\}$ and both its direct submultisets and its direct supermultiset over $F_{\mathrm{ID}} = \{a, b\}$.

Recall that a lemma specifies a security property. For our purpose, a security protocol theory consists of a protocol specification (setup rule and protocol rules), additional modelling assumptions (model specification rules and axioms), and security properties (lemmas).

A *verification task* $t_{\mathrm{V}}$ describes a security protocol theory with a single lemma and a set of human-error rules. Verification tasks are specified in Tamarin files and identified through the lemma and a task identifier. The lemma expresses the security property we prove in isolation and the task identifier specifies the set of human errors the given protocol is analysed against. We denote the lemma associated with a verification task by $t_{\mathrm{L}}$, and its task identifier by $t_{\mathrm{ID}}$.

To analyse a protocol's strength with respect to a specific task identifier, the failure rule in the specification file of $t_{\mathrm{V}}$ has to include the Fail facts corresponding to $t_{\mathrm{ID}}$ in its conclusion. We define a protocol's robustness against combinations of human errors with respect to a given property.

**Definition 3.9.** Given verification task $t_{\mathrm{V}}$. Let $P$ be the protocol described by $t_{\mathrm{V}}$ without human errors, and let $P'$ be the protocol described by $t_{\mathrm{V}}$ with human errors. That is, $P'$ is $P$ extended with one Fail fact for every failure identifier in $t_{\mathrm{ID}}$. The protocol $P$ is $t_{\mathrm{ID}}$-robust with respect to $t_{\mathrm{L}}$ if the protocol $P'$ provides the property expressed by $t_{\mathrm{L}}$.

### 3.2.1 Version without Helper Lemmas

For each verification task $t_{\mathrm{V}}$, there are three different possible results when we analyse the task with Tamarin:

**Verified.** The task's lemma $t_\mathrm{L}$ is verified for the given set of human errors.

**Timeout.** The proof search does not finish within the given amount of time and we do not know whether or not the property holds for the protocol.

**Falsified.** TAMARIN finds a counterexample and thus, the stated property does not hold for all protocol instances for the given human errors.

We introduced a way to graphically represent a set of task identifiers above in Figure 3.2. The algorithm is based on the idea to analyse the verification tasks in the same order a breadth-first search starting at the empty multiset visits the graph's nodes. This approach implements the concept to start the analysis at an infallible human and from there examine a skilled human. That is, we first analyse the protocol without errors and incrementally add more and more errors.

There are two special cases where we analyse a specific lemma only for an infallible human and not further.

- All lemmas we cannot verify for an infallible human are not of interest to our algorithm, as they cannot hold in the presence of human errors either. Thus, no further analysis is performed.
- The example trace of an `exists-trace` lemma is a valid proof for any number of failures, because we do not remove rules from the protocol specification. More precisely, we only add Fail facts to the failure rule's conclusion. Therefore, we do not need to repeat the verification for each combination of errors and know that the lemma always holds.

In the following, we consider a single lemma with an `all-traces` quantifier that was verified for an infallible human. For a fixed lemma in a protocol theory every verification task is uniquely identified by its task identifier. For the sake of convenience, we use the terms of a task and its identifier interchangeably. The actual search follows the steps below that describe how the next task is selected. For every task $t_\mathrm{V}$ selected for subsequent analysis, we store its task identifier $t_\mathrm{ID}$ in a queue.

1. By default, start with an infallible human. Therefore, initially add the empty multiset to the queue of task identifiers. If a positive value for the minimum number of failures is provided, generate and enqueue all task identifiers over $F_\mathrm{ID}$ with the specified number of failures.
2. Dequeue the next element $t_\mathrm{ID}$ and check whether any task corresponding to a submultiset of $t_\mathrm{ID}$ was previously falsified.
    - If this is the case, there is no need to analyse task $t_\mathrm{V}$ or any task corresponding to a supermultiset of $t_\mathrm{ID}$ any further.

    - Otherwise, analyse the verification task corresponding to $t_{\mathrm{ID}}$ with TAMARIN, terminate TAMARIN if the proof search does not find a proof within the specified amount of time, and store the outcome, i.e., *Verified*, *Timeout*, or *Falsified*. Add all direct supermultisets of $t_{\mathrm{ID}}$ to the queue if the outcome is either *Verified* or *Timeout*.

3. Repeat step 2 until the queue is empty or until the next task has more failures than the specified maximum number of failures.

We exploit that there is no need to analyse a specific task if another task allowing less errors was already falsified. Similar to an example trace of an `exists-trace` lemma, the attack trace of verification task $t_{\mathrm{V}}$ is a valid proof for lemma $t_{\mathrm{L}}$ with respect to task identifiers that are supermultisets of $t_{\mathrm{ID}}$. No matter how many `Fail` facts we add to the protocol, we can still find the same attack.

### 3.2.2 Extended Version with Helper Lemmas

We slightly extend the algorithm from the previous section to support helper lemmas in our analysis. Recall that helper lemmas may be used in the proofs of non-helper lemmas, and thus, have to be analysed when a non-helper lemma is verified. More precisely, a `reuse` lemma is used in the proofs of all lemmas that syntactically follow it and all `typing` lemmas are used in the proofs of all non-typing lemmas. Let $l_{\mathrm{H}}$ denote the set of helper lemmas that lemma $t_{\mathrm{L}}$ may depend on.

For each verification task $t_{\mathrm{V}}$, there are four different possible results when we analyse the task with TAMARIN:

**Verified.** The task's lemma $t_{\mathrm{L}}$ is verified for the given set of human errors and all helper lemmas in $l_{\mathrm{H}}$ are also verified.

**Maybe.** TAMARIN verifies the lemma $t_{\mathrm{L}}$, but some helper lemmas in $l_{\mathrm{H}}$ are not verified. Therefore, we do not know whether $t_{\mathrm{L}}$'s proof of correctness is valid.

**Timeout.** The proof search does not finish within the given amount of time and we do not know whether or not the property holds for the protocol.

**Falsified.** TAMARIN finds a counterexample and thus, the stated property does not hold for all protocol instances for the given human errors.

The order in which tasks are analysed is the same. However, helper lemmas, i.e., `reuse` and `typing` lemmas, are not analysed for a skilled human on their own. Instead, they are analysed together with the non-helper lemmas that require the proof of one or more helper lemmas. Therefore, we distinguish between the two outcomes *Verified* and *Maybe*. Again, non-helper lemmas whose outcome is not *Verified* for an infallible human are not analysed any further.

Recall the search steps for a fixed lemma $t_L$ in a protocol theory. When the lemma is verified for a specific task identifier $t_{ID}$, we additionally analyse all helper lemmas in $l_H$ against this combination of human errors. More precisely, we analyse all `reuse` lemmas syntactically preceding $t_L$ and all `typing` lemmas. Some `reuse` lemmas themselves may also depend on other helper lemmas which we check anyway. The proof of lemma $t_L$ is only valid after all helper lemmas in $l_H$ have been verified, denoted by *Verified*. Otherwise, we do not know whether the proof of $t_L$ is valid and denote such an uncertain outcome by *Maybe*.

## 3.3 Results Interpretation

The output directory generated by the tool has the following components:

- The `in` directory contains one input file for each verification task.
- All output files from TAMARIN are located in the `out` directory.
- The `summary.s` file stores the result for each verification task.

We produce one TAMARIN specification file for each analysed task and TAMARIN generates one output file for each proof search. Output files contain proofs for completed tasks and are empty if TAMARIN cannot find a proof within the given amount of time. We can directly load these output files in both the automated and the interactive mode to explore proofs and attacks.

The summary maps all analysed verification tasks to one of the possible results. *Maybe* results further have a list of helper lemmas and their results attached. For every lemma, the task identifiers are sorted alphabetically by the number of failures. All tasks that are within the specified bounds, but are not present in the summary, were indirectly falsified. To be exact, assume verification task $t_V$ is not in the summary and $|t_{ID}|$ is within the specified minimum and maximum number of failures. Then, there is a task $t'_V$ such that $t'_{ID} \subseteq^\sharp t_{ID}$ and TAMARIN falsified $t'_V$. Thus, we know that $t_L$ does not hold.

We can rerun the analyser several times to complete *Timeout* results after one run. The tool parses the output files generated by TAMARIN during previous runs and uses those results instead of calling TAMARIN again. This approach has several benefits: We can adjust the timeout over several runs, expand the set of human errors, analyse additional lemmas, call TAMARIN with an oracle during some runs, and introduce `reuse` and `typing` lemmas to find proofs more easily.

Users who run our tool multiple times on the same protocol should be aware that output files in the `out` directory are reused, while the specification files in the `in` directory and the summary are overwritten.

# 4 Case Studies

We analyse three communication protocols with our tool, run on a 3.40 GHz Intel i7 Quad-Core processor. We schedule a timeout of 600 seconds for an individual run of TAMARIN and restrict the number of failures by an upper bound of 4. All TAMARIN models are available at [4].

We model all protocols with respect to the human-error model introduced in Section 2.3. In addition to the failure identifiers summarised in Table 2.1, we introduce the new failure identifiers *kcc*, *ksc*, and *kgc*. The complete set of all failure identifiers we consider in our case studies is given in Table 4.1.

| $f_{\text{ID}} \in F_{\text{ID}}$ | Description |
|---|---|
| *msc* | Message sending confusion. |
| *csc* | Channel sending confusion. |
| *mrc* | Message receiving confusion. |
| *nlc* | No learning confusion. |
| *crc* | Channel receiving confusion. |
| *icc* | Ignoring comparison confusion. |
| *wrc* | Weak randomness confusion. |
| *soc* | Step order confusion. |
| *src* | Step repetition confusion. |
| *kcc* | Knowledge concatenation confusion. |
| *ksc* | Knowledge selection confusion. |
| *kgc* | Knowledge generation confusion. |

Table 4.1: Failure identifiers in $F_{\text{ID}}$ used for the case studies.

We derive the additional human-error rules from Schläpfer's human-knowledge rules [5]. Knowledge concatenation confusion, knowledge selection confusion, and knowledge generation confusion model a human who combines arbitrary terms in his knowledge, i.e., he may concatenate the wrong terms, concatenate the terms in a wrong order, or use any known term in place of a specified one. Users who fail to handle their knowledge and confuse messages can send arbitrary terms instead of the stated one. We extend the human-error model with the rules for knowledge confusion to cover such human errors and also control them explicitly. They are modelled by the rewriting rules explained next.

The rewriting rule for knowledge concatenation confusion is identified by the failure identifier *kcc*. It allows a human agent `A` to concatenate the two terms `m1` and `m2` from his knowledge.

```
/* Knowledge concatenation confusion. */
rule E_kcc:
   [ Fail($A,'kcc'), !HK($A,m1), !HK($A,m2) ]
   --[ Failed($A) ]->
   [ !HK($A,<m1,m2>) ]
```

The following human-error rule models knowledge selection confusion with failure identifier *ksc*. Human agent `A` selects both `m1` and `m2` from a pair of terms.

```
/* Knowledge selection confusion. */
rule E_ksc:
   [ Fail($A,'ksc'), !HK($A,<m1,m2>) ]
   --[ Failed($A) ]->
   [ !HK($A,m1), !HK($A,m2) ]
```

The failure identifier *kgc* represents knowledge generation confusion. It is modelled with the next human-error rule where human agent `A` can learn every public term.

```
/* Knowledge generation confusion. */
rule E_kgc:
   [ Fail($A,'kgc') ]
   --[ Failed($A) ]->
   [ !HK($A,$m) ]
```

In the following analysis, we consider the security properties entity and message authentication, confidentiality, and verifiability. Entity authentication states that an agent $S$ can be sure that another agent $H$ recently participated in the protocol. The property is more commonly known as *recent aliveness*.

**Definition 4.1.** [3] *Entity authentication* of an agent $H$ to another agent $S$ holds if whenever $S$ commits to $H$, $H$ has taken some action between the start of $S$ and the claim.

Message authentication states that an agent $S$ can be sure that a message $m$ was sent by another agent $H$, i.e., $H$ sends $m$ over an authentic channel to $S$.

**Definition 4.2.** [3] *Message authentication* of message $m$ from an agent $H$ towards another agent $S$ holds if whenever $S$ commits to $H$ and a message $m$, then $H$ has previously sent $m$.

Confidentiality states that an agent $H$ sends a message $m$ over a confidential channel to another agent $S$.

**Definition 4.3.** [5] *Confidentiality* of message $m$ holds when the adversary does not learn the specified message $m$.

Verifiability ensures the sender that the recipient actually received the message $m$.

**Definition 4.4.** [5] *Verifiability* for an agent $H$ with respect to a message $m$ and another agent $S$ holds if $H$ communicated $m$ to $S$ and may verify that $S$ learned $m$.

Next, we describe all three protocols, illustrate what properties are known to hold, and demonstrate what we found out about the protocols in our analysis. An overview of the obtained results for single human errors is given in Table 4.2. The complete results are available at [4], i.e., the results from thousands of tasks including single human errors and all combinations of up to 4 failures.

| | MP-Auth | Google | Helbach | | |
|---|---|---|---|---|---|
| | *EA* | *MA* | *C* | *MA* | *V* |
| *msc* | timeout | ✓ | × | ✓* | × |
| *csc* | timeout | ✓ | ✓ | ✓ | ✓ |
| *mrc* | ✓ | ✓ | ✓ | ✓ | ✓ |
| *nlc* | ✓ | ✓ | ✓ | ✓ | ✓ |
| *crc* | ✓ | ✓ | ✓ | ✓ | ✓ |
| *icc* | ✓ | × | ✓* | ✓* | ×* |
| *wrc* | ✓ | ✓ | ✓ | ✓ | ✓ |
| *soc* | ✓ | × | ✓ | ✓ | ✓ |
| *src* | ✓ | ✓ | ✓ | ✓ | ✓ |
| *kcc* | ✓ | ✓ | ✓ | ✓ | ✓ |
| *ksc* | ✓ | ✓ | ✓ | ✓ | ✓ |
| *kgc* | ✓ | ✓ | ✓ | ✓ | ✓ |

* result found in [5]

Table 4.2: Entity authentication (*EA*), message authentication (*MA*), confidentiality (*C*), and verifiability (*V*) of human $H$ to server $S$ for a single human error in the skilled human error class.

## 4.1 MP-Auth Protocol

MP-Auth is an authentication protocol to authenticate the human user $H$ to a remote server $S$. For this purpose, $H$ uses his personal device $D$ and his platform $P$. We assume $D$ to be an honest device and $P$ to be malicious, i.e., $P$ can be controlled by the adversary. To model a malicious platform $P$

$$
\begin{array}{rrl}
0. & \mathrm{H}: & \mathrm{knows}(D, P, S, pw, idH, idS) \\
0. & \mathrm{D}: & \mathrm{knows}(H, idH, \mathrm{pk}(skS)) \\
0. & \mathrm{S}: & \mathrm{knows}(skS, H, idS, pw, idH) \\
1. & \mathrm{H}\ \mathrm{h}\!\rightarrow\!\mathrm{h}\ \mathrm{P}: & S \\
2. & \mathrm{P}\ \circ\!\!\rightarrow\!\!\bullet\ \mathrm{S}: & `start' \\
3. & \mathrm{S}\ \bullet\!\!\rightarrow\!\!\bullet\ \mathrm{P}: & \mathrm{fresh}(rS).idS, rS \\
4. & \mathrm{P}\ \bullet\!\!\rightarrow\!\!\bullet\ \mathrm{D}: & idS, rS \\
5. & \mathrm{D}\ \mathrm{h}\!\rightarrow\!\mathrm{h}\ \mathrm{H}: & idS\ /\ identity \\
6. & \mathrm{H}: & \mathrm{compares}(identity, idS) \\
7. & \mathrm{H}\ \mathrm{h}\!\rightarrow\!\mathrm{h}\ \mathrm{D}: & pw, idH \\
8. & \mathrm{D}\ \bullet\!\!\rightarrow\!\!\bullet\ \mathrm{P}: & \mathrm{fresh}(rD).\mathrm{aenc}(rD, \mathrm{pk}(skS)), \\
 & & \mathrm{senc}(\langle f(rS), idH, pw\rangle, f(rS, rD))\ /\ forS \\
9. & \mathrm{P}\ \bullet\!\!\rightarrow\!\!\bullet\ \mathrm{S}: & forS\ /\ \mathrm{aenc}(rD, \mathrm{pk}(skS)), \\
 & & \mathrm{senc}(\langle f(rS), idH, pw\rangle, f(rS, rD)) \\
10. & \mathrm{S}\ \bullet\!\!\rightarrow\!\!\bullet\ \mathrm{P}: & \mathrm{senc}(f(rD), f(rS, rD))\ /\ forD \\
11. & \mathrm{P}\ \bullet\!\!\rightarrow\!\!\bullet\ \mathrm{D}: & forD\ /\ \mathrm{senc}(f(rD), f(rS, rD)) \\
12. & \mathrm{D}\ \mathrm{h}\!\rightarrow\!\mathrm{h}\ \mathrm{H}: & `success'
\end{array}
$$

Figure 4.1: Protocol MP-Auth [3].

in Tamarin, we make all channels to and from $P$ insecure and omit the role specification of $P$. The human authenticates himself with his password and his device. The goal of this protocol is to provide entity authentication from the human $H$ to the remote server $S$.

The detailed steps of the protocol are given in Figure 4.1. In Step 1, $H$ enters the server's name on $P$ which starts the communication with $S$ in Step 2. Then, $S$ sends its identity $idS$ and a freshly generated value $rS$ over $P$ to $D$ (Steps 3 and 4). In Step 5, $idS$ is displayed to the human who compares the displayed identity $identity$ with the identity $idS$ in his initial knowledge in Step 6. If they match, $H$ enters his identity $idH$ and his password $pw$ on $D$ (Step 7). Next, $D$ generates a fresh value $rD$ and encrypts it with the server's public key. Then, $D$ encrypts $\langle f(rS), idH, pw\rangle$ with $f(rS, rD)$ and sends both encrypted messages over $P$ to $S$ (Steps 8 and 9). Next, $S$ encrypts $f(rD)$ with $f(rS, rD)$ and sends it to $P$ in Step 10. $P$ forwards the message to $D$ which notifies the human about a successful authentication (Steps 11 and 12).

The authentication protocol MP-Auth was already analysed for human errors in the rule-based human error class in [3]. They showed that the protocol provides entity authentication from $H$ to $S$ for an infallible human, but not for an untrained human. In addition, it provides entity authentication for a rule-based human who follows the guideline to only enter his password on the device.

$$
\begin{array}{lrl}
0. & \mathrm{H}: & \mathrm{knows}(D, P, S, pw, m, idH) \\
0. & \mathrm{D}: & \mathrm{knows}(H) \\
0. & \mathrm{S}: & \mathrm{knows}(H, D, pw, idH) \\
1. & \mathrm{H} \ \mathrm{h}\!\!\twoheadrightarrow\!\!\mathrm{h} \ \mathrm{P}: & S, idH, pw, m \\
2. & \mathrm{P} \ \circ\!\!\twoheadrightarrow\!\!\bullet \ \mathrm{S}: & idH, m \\
3. & \mathrm{S} \ \circ\!\!\twoheadrightarrow\!\!\bullet \ \mathrm{D}: & \mathrm{fresh}(c).c, m \\
4. & \mathrm{D} \ \mathrm{h}\!\!\twoheadrightarrow\!\!\mathrm{h} \ \mathrm{H}: & c, m \ / \ c, message \\
5. & \mathrm{H}: & \mathrm{compares}(message, m) \\
6. & \mathrm{H} \ \mathrm{h}\!\!\twoheadrightarrow\!\!\mathrm{h} \ \mathrm{P}: & S, c \\
7. & \mathrm{P} \ \bullet\!\!\twoheadrightarrow\!\!\bullet \ \mathrm{S}: & c, pw, m
\end{array}
$$

Figure 4.2: Protocol Google 2-Step [3].

**Entity Authentication.** We verify MP-Auth's robustness against most single human errors with respect to entity authentication. Unfortunately, TAMARIN cannot find a proof of correctness or an attack trace in the given amount of time for the failure identifiers *msc* and *csc* as shown in Table 4.2.

However, there is an attack on the protocol when the user enters his password on the platform [3]. Thus, we would expect that the protocol is not robust against message sending confusion, i.e., not $\{msc\}$-robust, with respect to entity authentication.

Overall, TAMARIN can neither verify nor falsify many verification tasks within the specified amount of time, which demonstrates the need to rerun our tool multiple time with various input parameters. Moreover, TAMARIN's proof search can be supported with different heuristics and helper lemmas.

## 4.2 Google 2-Step Protocol

The Google 2-Step protocol also authenticates a human $H$ to a remote server $S$. As in Section 4.1, we assume that $H$ uses his trusted device $D$ and his platform $P$, and that $P$ can be controlled by the adversary. The human authenticates himself with his password, receives a fresh code on the device over a second channel, and has to enter this code on the platform. The goal of this protocol is to provide both entity and message authentication from $H$ to $S$.

Figure 4.2 depicts the protocol steps in extended Alice&Bob notation. In Google's two-factor authentication, $H$ enters his identity $idH$, his password $pw$, and the message $m$ on $P$ in Step 1. Next, $P$ contacts the server $S$ who generates a fresh code $c$ and sends both $c$ and $m$ to the human's device $D$ by text message (Steps 2 and 3). In Step 4, $D$ displays $c$ and $m$ to the human. Next, $H$ compares the displayed message $message$ with the message $m$ in his initial knowledge, and if they match, he enters the code on $P$ (Steps 5 and 6). Finally, $P$ sends the code, $pw$, and $m$ to $S$ in Step 7.

The authentication protocol Google 2-Step was already analysed for human errors in the rule-based human error class in [3]. They proved that the protocol provides entity authentication for both an infallible and an untrained human. Therefore, it is not worthwhile to test this property, as it holds for any number of errors the human can make. We examine the message authentication property instead. *Basin et al.* [3] showed that the protocol provides message authentication for an infallible human and a rule-based human who follows the guideline to always compare a received message with the one in his initial knowledge. They found an attack on the protocol with an untrained human when the user omits the comparison.

**Message Authentication.** We find the following known attack on the Google 2-Step protocol. The protocol is neither $\{icc\}$-robust nor $\{soc\}$-robust with respect to message authentication, i.e., the protocol is neither robust against ignoring comparison confusion nor against step order confusion. The Tamarin prover finds the same attack in either case. The adversary intercepts the communication of $m$ and forwards a freshly generated or public term to the server. Since both human-error rules allow the human user to not compare the received term with the initially sent message, the user erroneously proceeds with the authentication. Either the user does not compare the terms properly or skips the step completely. Therefore, message authentication does not hold, i.e., the server commits to the user $H$ and a message that was not sent by $H$. This can be interpreted the same as the attack found by [3].

The complete results at [4] show that message authentication holds for all other single human errors and many combinations of them. Our tool finds the attack we expected, but does not find any further attack.

## 4.3 Helbach Code Voting Variant

The Helbach code voting variant is an Internet voting protocol. It allows a human $H$ to submit his vote to a server $S$ using his platform $P$. We assume $P$ to be an honest platform. The voter receives a code sheet beforehand. It contains the following information for the candidates $c$ and $c'$:

- The voting codes denoted by $\mathrm{h}(c)$ and $\mathrm{h}(c')$.
- The verification codes denoted by $\mathrm{h}(\langle k, c\rangle)$ and $\mathrm{h}(\langle k, c'\rangle)$.
- The finalisation codes denoted by $\mathrm{h}(\langle k', c\rangle)$ and $\mathrm{h}(\langle k', c'\rangle)$.

We assume that the voter receives the code sheet over a secure channel and that he first reads all information from the code sheet at once. We model this information in the human's initial knowledge. The goal of this protocol

$$
\begin{array}{lrl}
0. & \mathrm{H}: & \mathrm{knows}(S, c, \mathrm{h}(c), \mathrm{h}(\langle k, c\rangle), \mathrm{h}(\langle k', c\rangle), c', \mathrm{h}(c'), \mathrm{h}(\langle k, c'\rangle), \mathrm{h}(\langle k', c'\rangle)) \\
0. & \mathrm{S}: & \mathrm{knows}(H, c, \mathrm{h}(c), \mathrm{h}(\langle k, c\rangle), \mathrm{h}(\langle k', c\rangle), c', \mathrm{h}(c'), \mathrm{h}(\langle k, c'\rangle), \mathrm{h}(\langle k', c'\rangle)) \\
1. & \mathrm{H} \ \mathrm{h}\!\rightarrow\!\mathrm{h} \ \mathrm{P}: & \mathrm{h}(c) \ / \ \mathit{code} \\
2. & \mathrm{P} \ \circ\!\rightarrow\!\circ \ \mathrm{S}: & \mathit{code} \ / \ \mathrm{h}(c) \\
3. & \mathrm{S} \ \circ\!\rightarrow\!\circ \ \mathrm{P}: & \mathrm{h}(\langle k, c\rangle) \ / \ \mathit{verificationcode} \\
4. & \mathrm{P} \ \mathrm{h}\!\rightarrow\!\mathrm{h} \ \mathrm{H}: & \mathit{verificationcode} \\
5. & \mathrm{H}: & \mathrm{compares}(\mathrm{verificationcode}, \mathrm{h}(\langle k, c\rangle)) \\
6. & \mathrm{H} \ \mathrm{h}\!\rightarrow\!\mathrm{h} \ \mathrm{P}: & \mathrm{h}(\langle k', c\rangle) \ / \ \mathit{finalisationcode} \\
7. & \mathrm{P} \ \circ\!\rightarrow\!\circ \ \mathrm{S}: & \mathit{finalisationcode} \ / \ \mathrm{h}(\langle k', c\rangle)
\end{array}
$$

Figure 4.3: Helbach code voting variant [5].

is to provide confidentiality, message authentication, and verifiability from the human $H$ to the remote server $S$.

The protocol is shown in Figure 4.3. First, the human enters the candidate's voting code on $P$ who sends the vote to $S$ (Steps 1 and 2). Next, $S$ sends the corresponding verification code over $P$ back to $H$ (Steps 3 and 4). In Step 5, $H$ compares the displayed verification code with the one he intended to send. If they match, he confirms the vote's correctness with the corresponding finalisation code (Steps 6 and 7).

The Helbach code voting variant was already analysed for human errors in the skilled human error class in [5]. They showed that the protocol provides confidentiality, message authentication, and verifiability for an infallible human. However, the Helbach code voting variant is not {*icc*}-robust with respect to the verifiability property, and not {*icc,msc,msc*}-robust with respect to message authentication.

**Confidentiality.** We prove that the Helbach code voting variant is not {*msc*}-robust with respect to the confidentiality property. Tamarin finds a new attack where the human user confuses messages and sends the candidate's information in the clear instead of the corresponding voting code. Hence, confidentiality does not hold.

Tamarin verifies the confidentiality property for all other single human errors and most combinations of errors.

**Message Authentication.** The Helbach code voting variant is both {*kcc*}-robust and {*msc*}-robust with respect to message authentication, but the errors lead to an attack in combination with each other, i.e., the protocol is not {*kcc,msc*}-robust. We find the new attack where the user first concatenates the voting code and the finalisation code. Then, he enters the pair on his platform instead of the voting code. The adversary intercepts the code pair and impersonates the human user. Thus, message authentication does not hold.

Message authentication holds for all single human errors. Furthermore, TAMARIN finds no other attack within the timing bounds, but we would expect to find an attack on the protocol for the task identifier $\{icc,msc,msc\}$. *Schläpfer* [5] showed that the Helbach code voting variant is not $\{icc,msc,msc\}$-robust with respect to message authentication.

**Verifiability.** We find the following new attack on the Helbach code voting variant. It is not $\{msc\}$-robust with respect to the verifiability property. TAMARIN finds an attack where the user sends the verification code to his platform instead of the voting code. Then, the adversary intercepts the verification code and returns it to the user's platform. The user compares the verification code and erroneously continues with the finalisation code, who never reaches the server. Therefore, verifiability does not hold.

Moreover, we find the following known attack on the Helbach code voting variant which is not $\{icc\}$-robust with respect to the verifiability property. TAMARIN finds an attack where the human user omits the comparison of the verification code. This allows the adversary to intercept the voting code and to return a freshly generated or public term. The user accepts this term and so, verifiability does not hold, as the server does not learn the candidate.

We find no other attack on the protocol and can verify the verifiability property for all other single human errors. The detailed results are available at [4].

# 5 Conclusion

We have introduced an algorithm that efficiently generates different combinations of human errors, and tests them with respect to a protocol's security properties. We implemented our algorithm in a tool to automatically analyse communication protocols with human errors. The tool uses Tamarin as a back-end to analyse the individual tasks. Finally, we performed three case studies to show the utility of our tool, and found several attacks on communication protocols with respect to different human errors. We were able to automate the time-consuming and repetitive process of specifying and analysing a new protocol theory for every set of human errors.

However, our approach is still limited in the sense that we only support the skilled human error class. An alternative approach is to assume the worst case possible with respect to the human's capabilities. That is, we start the analysis with an untrained human and further examine a rule-based human. Instead of human-error rules, we need a set of guidelines for a rule-based human, and then, control the number of active guidelines.

Another idea is to improve the presented algorithm for a skilled human. For this purpose, we could first test the tasks on the lower and the upper bound of the search space, and afterwards start an adjusted bisection method. This approach would additionally exploit that there is no need to analyse a task if another task allowing more errors was already verified.

Moreover, we could consider an infinite number of errors and accordingly check whether a lemma is robust against any number of particular failures. This could be implemented with the same human-error rules and persistent facts similar to the linear Fail facts used. To exploit their full potential, these persistent facts could be combined with each other just like we combine linear Fail facts.

In our current approach, the number of Tamarin runs is tremendous and as the Tamarin prover does not always terminate, this limits the tool's practicality. Therefore, future work should also examine how Tamarin could terminate faster.

In summary, our analyser is a work-saving tool to specify and analyse protocols with human errors using Tamarin. It systematically tests all desired combinations of human errors with respect to different trace properties.

# Bibliography

[1] Tamarin Prover. `http://www.infsec.ethz.ch/research/software/tamarin.html`.

[2] The Tamarin Prover Manual. `http://tamarin-prover.github.io/manual/index.html`.

[3] David Basin, Saša Radomirović, and Lara Schmid. Modeling Human Errors in Security Protocols. In *Computer Security Foundations Symposium (CSF), 2016 IEEE 29th*, pages 325–340, 2016.

[4] Andrina Denzler. Tool Source Code and Case Study Specification Files. `http://www.infsec.ethz.ch/research/projects/hisp.html`.

[5] Michael Schläpfer. *Secure End-To-End Communication in Remote Internet Voting*. PhD thesis, ETH Zurich, 2016.

[6] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. Automated Analysis of Diffie–Hellman Protocols and Advanced Security Properties. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 78–94, 2012.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

## Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

---

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

**Title of work** (in block letters):

Automatic Analysis of Communication Protocols with Human Errors

**Authored by** (in block letters):
*For papers written by groups the names of all authors are required.*

| **Name(s):** | **First name(s):** |
|---|---|
| Denzler | Andrina |
| | |
| | |
| | |
| | |

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

| **Place, date** | **Signature(s)** |
|---|---|
| Nänikon, 07.10.2016 | ADenzler |
| | |
| | |
| | |
| | |

*For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.*