E-voting and secure human-server communication

Master Thesis

Author(s):
Zgraggen, Tomas

Publication date:
2014

Permanent link:
https://doi.org/10.3929/ethz-a-010255825

E-Voting and Secure Human-Server Communication

Author: Tomas Zgraggen
Supervisor: Saša Radomirović
Professor: David Basin
Institute of Information Security, ETH Zurich, Switzerland

Issue Date: March 5, 2014 – Submission Date: September 4, 2014

Abstract

Security-relevant protocols often concern the communication between a human user and a server. Models of these protocols are generally imprecise; for instance, they do not include the human as an agent in the model. Some of these issues can be addressed by using the HISP model, which attempts to offer a more realistic underlying system of human-server communication. The HISP model is currently missing important features such as strong authentication properties. In this master's thesis, we improve the HISP model by extending it with additional channel properties such as replay-free channels for the interaction between humans and devices, as well as security properties such as injective agreement. We characterize the HISP topologies using these new properties, and apply our modifications and our insights to model and verify a draft proposal for a Swiss E-voting protocol. We discuss a flaw found in this protocol and suggest a possible fix.

Contents

1 Introduction

Many security-relevant protocols used today concern the communication of a human user with some remote server. Examples include protocols for E-Banking, E-Voting, online shopping, or simply securely accessing social media accounts. To verify the security of these protocols, model checkers such as Scyther [1] or ProVerif [2] are used to create and reason about models of these protocols. However, there are no guidelines as to how the protocols or the properties to be proven should be expressed. In particular, the human is mostly not considered to be an actor in the protocol, and there is no consensus on how the underlying system of humans and computer agents should be modeled.

The Human Interaction Security Protocol (or HISP) model, utilizing the automated proof checker Tamarin [3], provides a formal way of identifying the participants in these protocols and how they interact with each other. It introduces the human as an actor with limited computing capabilities reliant on external devices to perform cryptographic operations, as well as taking into account that some of these devices may be compromised by an adversary. Additionally, the HISP model allows us to express security properties (such as confidentiality) as channel assumptions as well as protocol goals. This allows us to simply define a channel as having a certain property, rather than having to ensure this property in the model of the protocol. Since the HISP model is new, there is still plenty of room for expansion and improvement. This thesis focuses on expanding the current HISP model with additional channel and security properties, as well as applying these additions in a case study.

Contributions
We introduce novel channel properties in addition to the already defined properties of authenticity, confidentiality and security. In particular, we add the ability for channels to be replay-free, available, and immediate (in the sense that no protocol rule can be executed while a message in this channel has not yet been received). We define these additions in a modular way, allowing for arbitrary combinations of the different properties to create close approximations of real-world channels.

We also add additional security properties based on Lowe's authentication properties [4]. Relative to the current authentication property provided by the HISP model, we include weaker properties such as aliveness and recentness, as well the stronger properties of injective and non-injective agreement and injective and non-injective synchronization.

Using the non-injective agreement property, we perform a characterization of all HISP topologies described in [5]. Furthermore, to verify the injective agreement property, we extend all protocol models and corresponding HISP topologies. In a final step, we use the gained knowledge as well as

the newly created channel and security properties to model, verify and fix a proposed protocol for the Swiss E-Voting system.

Organization
We give an introduction to the HISP model and the Tamarin prover in Section 2. Section 3 focuses on the description of the extended channel properties, while Section 4 describes the various security properties. We characterize the existing HISP topologies with regard to the new security properties in Section 5. Finally, we utilize our previous work in a case study (a proposed E-Voting protocol) in Section 6 and present our conclusions in Section 7.

2 Background

2.1 The HISP Model

The HISP model was created by Michael Schläpfer and Saša Radomirović to better describe and formalize communication between a human and a remote communication partner. It consists of a graph-theoretical model called a *communication topology* on top of an extended protocol model. This model can be used to characterize necessary and sufficient conditions for the existence of security protocols that provide secure channels between a human and a remote server. The paper detailing this model in full can be found at [5]. In the following, we outline the basic concepts of the topology and the model extension necessary for this thesis.

2.1.1 The Topology Model

As stated before, the HISP topologies are graph-theoretic models and thus usually represented as a graph. A HISP topology consists of four nodes, representing the roles specified in a protocol. These are:

- The human, denoted by H. In order to realistically model the average human, the authors explicitly disallow the human from performing any cryptographic operations, as well as requiring the human's initial knowledge to be empty.
- A platform, usually a computer or other medium the human uses to connect to the internet, denoted by P. This platform can be compromised and untrustworthy, modeling the real-world possibility of a computer infected with malware.
- A device D. This device is assumed to be trusted, and can be anything from a TPM to a piece of paper containing codes.
- The server S. This is the remote partner communicating with the human.

Any channels between these actors are represented as directed edges, forming a graph. These channels can have different properties, such as being confidential or authentic, rather than just simply being insecure.

The supergraph of all possible HISP topologies can be seen in Figure 1, where we have secure channels from H to D and D to S (and vice versa), as well as insecure channels between P and all other nodes. All honest roles are represented by a regular circle. We use a dashed circle to represent the honest, but limited role H. The two concentric circles are used to represent the unlimited but

dishonest role P. The symbols next to the edges represent the type of channel connecting the two nodes, with $\circ\!\!\rightarrow\!\!\circ$ being insecure channels, $\bullet\!\!\rightarrow\!\!\circ$ being authentic channels, $\circ\!\!\rightarrow\!\!\bullet$ being confidential channels and $\bullet\!\!\rightarrow\!\!\bullet$ being secure channels.

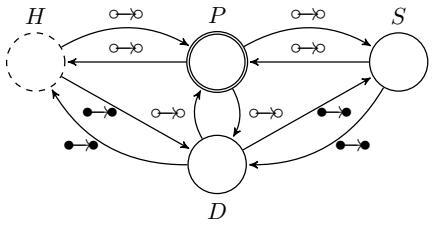


Figure 1: The supergraph of all HISP topologies.

Note that in the following, we use the functions $\langle \_, \_ \rangle$ for pairing, $\pi_1(\_)$ and $\pi_2(\_)$ for the first and second projection of a pair of terms, as well as $\mathcal{C}_{\mathsf{pub}}$ and $\mathcal{C}_{\mathrm{fresh}}$, denoting the two countably infinite, disjoint sets of fresh and public constants.

Formally, [5] defines a HISP topology as follows:

Definition 1. *A* HISP topology *is a topology* (V, E, η, μ)*, where the set of roles is* $V = \{H, D, S, P\}$ *and the set of links is* $E \subseteq \{(H,P),(P,H),(D,P),(P,D),(P,S),(S,P),(H,D),(D,H),(D,S),(S,D)\}$.

The vertex labeling is given by $\eta(H) = (\Sigma_H, \emptyset, \mathsf{honest})$*,* $\eta(D) = (\Sigma, \mathcal{T}, \mathsf{honest})$*,* $\eta(S) = (\Sigma, \mathcal{T}, \mathsf{honest})$*, and* $\eta(P) = (\Sigma, \mathcal{T}, \mathsf{dishonest})$*, where* $\Sigma_H = \{\langle \_, \_ \rangle, \pi_1(\_), \pi_2(\_)\} \cup \mathcal{C}_{\mathsf{pub}} \cup \mathcal{C}_{\mathrm{fresh}}$.

The edge labeling is given by $\mu(e) = \circ\!\!\rightarrow\!\!\circ$*, for* $e \in E_1 \cap E$*, and* $\mu(e) = \bullet\!\!\rightarrow\!\!\bullet$*, for* $e \in E_2 \cap E$*, where* $E_1 = \{(H,P),(P,H),(D,P),(P,D),(P,S),(S,P)\}$ *and* $E_2 = \{(H,D),(D,H),(D,S),(S,D)\}$.

2.1.2 Protocol Model Extensions

The authors describe several extensions to the protocol model, which are discussed in detail in Section 2.2. In the following, we provide an overview of the extensions to the model:

Dishonest Agents
In general, the adversary is modeled as a part of the underlying (insecure) communication network, and thus can only gain information or modify messages that are sent through this network. Because the HISP protocol model explicitly models the agent state, the adversary can be allowed to corrupt an agent, and can perform operations such as reading out or modifying the agent state, as well as choosing compromised "random" values for the agent. Dishonest agents are marked as such in the model and cannot return to an honest state.

Channel Assumptions
As stated before, the general assumption about the network in protocol models is that it is insecure, with an adversary being able to intercept and modify messages sent through it. It is the responsibility of the model author to ensure confidentiality and authenticity properties of the protocol through the use of cryptographic functions and nonces. This assumption does not however reflect real world channels perfectly. For example, we might want to simply assume that the communication between a user and a trusted device in his possession is already secure. The extended model

allows us to use channels with additional properties, such as authenticity, without needing to model these properties in the protocol.

Channels as Goals
The purpose of the HISP model is to reason about the existence of protocols providing certain channel properties. This is done by requiring the existence of a protocol which not only fulfills the security properties, but also communicates a message from a sender to a receiver (thus preventing secure, but useless protocols). Note that communicating a message is a more general action than simply transmitting a message. For example, in code voting, a user transmits a code but communicates the name of the candidate associated with that code.

2.2 The Tamarin Prover

The Tamarin prover [3] is a security protocol verification tool that supports falsification and unbounded verification of security protocols. It functions in a similar way to other proof checkers such as ProVerif [2] or Scyther [1], but offers more functionality, such as the ability to model exponentiation (used e.g. for Diffie-Hellman Key Exchange). The Tamarin code repository can be found at [7].

Protocols in Tamarin are described as multiset term rewriting systems, and the system states are represented as finite multisets of facts (with the initial system state being the empty multiset). These facts are used in labeled multiset rewriting rules which represent state transitions. Such a rule consists of premises, actions, and conclusions, as shown here:

$$[\mathsf{Premise}] -[\ \mathsf{Action}\ ]\!\!\rightarrow [\mathsf{Conclusion}]$$

This rule rewrites the current state by replacing the linear facts in premise with the facts in the conclusion and is labeled with the action Action. Note that all facts in the premise of the rule must occur in the state for that rule to be applicable.

In the following, we will cover the basics required to read and understand the protocol models presented throughout this thesis. For a more complete description of Tamarin, consult [3].

2.2.1 Basics

A Tamarin protocol model is generally split into two parts, one dealing with modeling the actual protocol via protocol rules, the other dealing with various security properties that the protocol should fulfill. These rules are explained in the following sections, but first we provide some information on the various mechanisms of Tamarin.

Functions and Equations
Tamarin allows for the modeling of custom functions. The following example defines three functions, a hash function taking two parameters (e.g. a message and a key), and two functions each taking one parameter.

```
functions: h/2, f/1, g/1
```

Unless an equational theory is specified, all these functions are hash functions, since there is no way for Tamarin to regain a message once a function has been applied to it. To do this, we can define equations for individual functions. The following example shows how f and g are defined as inverse functions:

```
equations:  f(g(m)) = m, g(f(m)) = m
```

Tamarin can now use these equations to restore a message m once a function f or g has been applied to it. Of course, it is not necessary to manually define common functions such as encryption or hashing. These functions can be included by using the builtins keyword. For example, including the symmetric encryption/decryption function senc can be done via:

```
builtins:  symmetric-encryption
```

In this thesis we will be using one specific builtin module called multiset. This module provides an operator "+" which allows us to express multiset union. This allows us, for instance, to create sets of messages simulating a counter by adding additional messages into the set.

Message Types
Tamarin supports three different message types. These are:

- Public Messages: These messages are denoted either by a preceding "$" symbol for variables or by enclosing alphanumeric strings with single quotation marks for constants (e.g. $'const'$). These messages are known to all parties (including the adversary) a priori and without needing to explicitly add them to a party's knowledge.
- Fresh Messages: These messages are generated exclusively by the builtin Fresh rule. A fresh message variable is denoted by a preceding "~" symbol. The Fresh rule is guaranteed to never instantiate two variables with the same value.
- Regular Messages: These messages have no preceding symbol. Regular messages are a supertype of both fresh and public messages, meaning a regular message variable can be instantiated with a fresh or public value (but not vice versa).

2.2.2 Protocol Rules

The protocol rules model the actual protocol and make up the main part of every model. In the following, we will detail how rules are created and how they interact with themselves and the adversary.

Rule Creation
A generic rule has the following structure:

$$[\mathsf{Premise1}(\ldots), \mathsf{Premise2}(\ldots), \ldots] -[\ \mathsf{Action1}(\ldots), \ldots\ ]\rightarrow [\mathsf{Conclusion1}(\ldots), \ldots] \tag{1}$$

Protocol rules cause state transitions by replacing facts in the state of the system and consist of three parts, namely premises, actions, and conclusions. In order to apply a rule, all facts listed in the premise of that rule must be in the state, i.e. they must have been created at some point prior to the application of the rule. After the rule is applied, the facts in the premise are removed from and the facts in the conclusion are added to the state of the system.

Facts as shown in the premise of Rule 1 are consumed upon use, and can hence only be used once. These facts are called *linear*. In addition to these, there are *persistent* facts, which can be used an unlimited number of times and are denoted by a preceding "!" symbol.

Actions are used to track events in the trace. Actions are logged when the rule executes and are used in lemmas and axioms to verify (or falsify) certain security properties. Each action is assigned a timestamp at the execution of the rule, so it is possible to reason about temporal order in the lemmas.

Rule Interaction and the Adversary
The initial state of the system is the empty multiset. This means that only rules which have an empty premise, such as the Fresh rule, may be executed. These rules will generate facts in their conclusions, which can then be used to execute other rules. Rules are executed based on pattern matching. This means that a fact must have the same name, number of arguments, and type of arguments (minus the substitutions for different message types) for it to be accepted as a valid premise for a rule.

The "adversary" is modeled by a separate set of rules, such as a fresh rule (lowercase) to create his own fresh values. The adversarial knowledge is a separate set which is originally empty. Tamarin is allowed to perform any builtin or custom defined functions on the elements of this set, such as pairing two elements or computing a hash function of an element, and may add these newly created values to the set. Additionally, new values can be added to this set by the Out fact. If this fact is present in the conclusion of a rule, its message term is added to the knowledge set of the adversary. Similarly, the adversary may submit any knowledge from the set into a protocol rule with the In fact. Note that it is not possible to transfer the contents of any other fact into the knowledge set of the adversary without a rule containing an Out fact.

Since we usually want to simulate an insecure network, such as the Internet, we model protocols in a way that all messages between parties are exchanged via In and Out facts. This means the adversary gains knowledge of the transmitted messages, as we would expect in an insecure network.

2.2.3 Security Properties

While protocol rules are used to express the model of the protocol, we now want to express properties of protocols such as authenticity and security. We do this by using the actions occurring in protocol rules in *lemmas* and *axioms*.

Temporal Logic Basics
Security properties are expressed as logical formulas in a guarded fragment of first-order logic which allows quantification over message and temporal variables. A security property is fulfilled if the formula evaluates to true. To create these formulas, Tamarin provides the logical operators "&" and "|", as well as the quantifiers All (∀) and Ex (∃).

To express that actions occur at certain times, Tamarin introduces the new variable type of temporal variables (denoted with a preceding "#" symbol), as well as the operator "@". For example, we could write:

```
Ex #i #j.  A1() @i => A2() @j & i<j
```

We have stated that there exist two points in time i and j such that there is an action A1 occurring at i and an action A2 occurring at j, and that i occurs before j. We can now use this syntax to formulate axioms and lemmas.

Axioms
Axioms are trace restrictions, expressing properties that the set of traces we analyse must have. Tamarin will compare each trace to the specified axioms, and discard it if an axiom is not fulfilled. To illustrate, assume that we have a certain protocol rule that we only want to execute once per trace. This rule might be written as:

$$[\,]\!-\![\ \mathsf{CheckAction}()\ ]\!\to[\mathsf{Conclusion1}(\ldots),\ldots]$$

Without an axiom, there is no way to prevent Tamarin from creating traces where this rule is executed more than once, since there are no premises in the rule and hence it can be executed an unlimited amount of times. To prevent this, we now create the following axiom:

```
axiom run_once:
"All #i #j.  CheckAction() @i & CheckAction() @j ==> #i=#j"
```

This axiom states that for all times i and j, it must hold that if CheckAction has been logged at i and at j, the two variables must be equal. Obviously, if there is a trace where CheckAction occurs multiple times, this property would be violated. Since Tamarin only keeps traces that fulfill all axioms, any trace calling the previous rule multiple times would be discarded.

Lemmas
Lemmas, as opposed to axioms, are used to express properties that the protocol should fulfill. Not fulfilling a lemma will cause Tamarin to report an attack.

Lemmas come in two variants. The first (and most commonly used) is the universal lemma. As the name suggests, the formula expressed by this lemma must hold for all valid traces (i.e., traces that fulfill all axioms). This type of lemma is used to express properties such as confidentiality or authenticity, since obviously such a property must hold for all traces. The following example shows a lemma describing confidentiality:

```
lemma confidential:
"All S R m #k.  Secret(S,R,m) @k ==> not (Ex #j.  K(m) @j)"
```

This lemma makes use of a special action K, which denotes the knowledge of the adversary. In words, this formula states that for all agents S and R and all messages m, if there is a secrecy claim Secret(S,R,m) at time k, then there must not exist a time j where the message m is contained in the adversary's knowledge set.

Obviously, care must be taken when writing these lemmas to cover any corner cases in the protocol model. For example, it might make sense to allow a dishonest agent to leak information to the adversary. A way to achieve this would be to amend the previous lemma to:

```
lemma confidential:
"All S R m #k.  Secret(S,R,m) @k ==> not (Ex #j.  K(m) @j) | (Ex #j.  Dishonest(S) @j
| Dishonest(R) @j)"
```

Here, we allow the adversary to learn the message m (i.e. we allow the first disjunct of the formula to be false), but only if either one of the involved agents is dishonest (this is done by an additional

protocol rule which logs the Dishonest action).

The second type of lemma is the existential lemma, which is denoted by the keyword exists − trace. For this lemma, it suffices that there is at least one trace that fulfills it. This type of lemma is usually used to guarantee that a protocol is not only secure, but also functional, i.e. there must be at least one trace where exchange of knowledge takes place. A simple example might be the following lemma:

```
lemma functional:  exists-trace
"Ex R m #i.  Message_Received($R,m) @i"
```

Here, we demand that there exists a trace where a Message_Received action has occurred, e.g. to indicate that there has been a transfer of knowledge from some sender to the receiver R.

2.3 Modeling HISP in Tamarin

Now that we have established a basic understanding of how protocols and properties are modeled in Tamarin, we discuss the modeling of HISP protocols.

2.3.1 Protocol Rules

The HISP protocol rules in general follow the premise established in the previous section, using In and Out facts to simulate an insecure network with an adversary. In the following, we list the additions to this basic premise.

Agent State
Since we want to model a continuous protocol rather than just disjointed rules, we need to establish a "flow" through the different rules. We do this by maintaining state, and requiring that each rule needs a certain state to be executed and in turn modifies the current state after execution. This is done by using linear AgentState facts which appear in the premise and conclusion of every protocol rule. We will illustrate the AgentState fact in the following example:

$$[\mathsf{Rcv}(\$S, \$R, m), \mathsf{AgentState}(\$R, 'R\_0', 'empty')] -[\ ]\rightarrow [\mathsf{AgentState}(\$R, 'R\_1', m)]$$

In this rule, R receives a message m from S. Let us now compare the two AgentState facts. The first parameter is the agent executing the rule. This value remains constant throughout the entire run.

The next parameter is the public constant "R_0", which is changed to "R_1" in the conclusion of the rule. This value stores the current position of the agent in the protocol, and is used to both ensure that a protocol rule cannot be repeated as well as to prevent an agent from executing protocol rules out of order.

The final parameter denotes the knowledge that the agent has accumulated. In the beginning, this set is empty, denoted by the public value $'empty'$. Once R has learned m, it this value is replaced. Additional values would be added to a set, e.g. adding a second message $m2$ would result in a set $\langle m, m2 \rangle$.

Setup
We set up HISP protocols via the Setup rule. This rule is responsible for assigning agents to roles and creating the initial knowledge for agents. Consider this example:

$$[\,]-[\ \mathsf{Setup}()\ ]\rightarrow[\mathsf{AgentState}(\$H,'H\_0','empty'),\mathsf{AgentState}(\$S,'S\_0',\$H)]$$

This setup rule assigns an agent to the role H (with no initial knowledge) and another to the role S (with the identity of his counterpart H included in the initial knowledge).

Dishonest Agents
The HISP model allows for agents to become dishonest, thereby leaking information to the adversary or accepting information from him. More precisely, we allow the adversary to do the following things:

- Learn the state of a dishonest agent. This is done with the rule:

$$[\mathsf{AgentState}(\$A,c,n)]-[\ \mathsf{Dishonest}(\$A)\ ]\rightarrow[\mathsf{Out}(\langle\$A,c,n\rangle)]$$

- Modify the state of a dishonest agent. Done with the rule:

$$[\mathsf{AgentState}(\$A,c1,n1),\mathsf{In}(\langle c,n\rangle)]-[\ \mathsf{Dishonest}(\$A)\ ]\rightarrow[\mathsf{AgentState}(\$A,c,n)]$$

- Manipulate the randomness of a dishonest agent (by corrupting any fresh values that this agent generates), with the rule:

$$[\mathsf{In}(r)]-[\ \mathsf{Dishonest}(\$A)\ ]\rightarrow[\mathsf{Fresh}(\$A,r)]$$

We log an agent becoming dishonest via the Dishonest action. We also require that an agent cannot become honest again in future runs once he has become dishonest. This is ensured by the following axiom:

```
honest_dishonest:
"All A #i #j.  Honest(A) @i & Dishonest(A) @j ==> F"
```

Note that we give the adversary more opportunities to corrupt an agent by splitting most rules into two parts. We do this for any rule that has a Snd as well as a Rcv fact. Consider the following example rule:

$$[\mathsf{AgentState}(\$H,'H\_0','empty'),\mathsf{Rcv\_S}(\$S,\$H,m)]-[\ ]\rightarrow$$
$$[\mathsf{AgentState}(\$H,'H\_1',m),\mathsf{Snd\_S}(\$H,\$D,m)]$$

We split this rule into the two following rules:

$$[\mathsf{AgentState}(\$H,'H\_0','empty'),\mathsf{Rcv\_S}(\$S,\$H,m)]-[\ ]\rightarrow[\mathsf{AgentState}(\$H,'H\_1',m)]$$
$$[\mathsf{AgentState}(\$H,'H\_1',m)]-[\ ]\rightarrow[\mathsf{AgentState}(\$H,'H\_2',m),\mathsf{Snd\_S}(\$H,\$D,m)]$$

This split allows us to model the adversary corrupting an agent while it is processing a message, rather than only allowing corruption before or after a message is sent or received.

Trust
In HISP models, we can express that a certain party (usually H) can trust another party (usually D). This is done with the action Trust($D). We can use this action to allow a lemma to verify even if a security property is not fulfilled, if a trusted party becomes dishonest (since we assume that this can never happen, we are not interested in the result of a trace if it does occur). The full confidentiality property using this action would be:

```
lemma confidential:
"All S R m #k. Secret(S,R,m) @k ==> not (Ex #j. K(m) @j) | (Ex #l. Dishonest(S) @l)
| (Ex #l. Dishonest(R) @l) | (Ex T #l. Trust(T) @k & Dishonest(T) @l)"
```

Functionality
As stated before, functionality is checked by an existence lemma. To check the functionality of HISP protocols, we first add two actions Comm and Learn to the model. The Comm action is used by the sender and contains as a parameter a message that he wants to communicate to the receiver. Similarly, the receiver will use the Learn action to log the information he has learned. We can now verify functionality with the following (simplified) lemma:

```
lemma functional: exists-trace
"Ex m S R #i #j. not S = R & Comm(S,m) @i & Learn(R,m )@j & i < j & not (Ex #l.
Dishonest(S) @l) & not (Ex #l. Dishonest(R) @l)"
```

2.3.2 Channel Rules

As discussed before, the general model for protocols in Tamarin is assuming an insecure network and exchanging messages via In and Out facts. However, this does not always model reality, and we want more choices of channel types in our model. To achieve this, HISP models communicate via channel rules. These will be discussed in more detail in Section 3.

We will however present the general concept here: Messages are passed into the network by protocol rules using Snd facts. These facts have an additional identifier, e.g. Snd_I, to determine which type of channel they are representing (in this case, an insecure channel). These facts are converted into corresponding Rcv facts by channel rules. These Rcv facts are then used as a premise in the protocol rules. We illustrate this by means of the following example, which shows the insecure channel. First, a message is sent with the rule shown below:

$$[\mathsf{Premise1}(m)] -[\ ]\!\rightarrow [\mathsf{Snd_I}(A, B, m)]$$

Next, the two following channel rules convert the Snd fact into a Rcv fact:

$$[\mathsf{Snd_I}(A, B, m)] -[\ \mathsf{Snd_I}(A, B, m)\ ]\!\rightarrow [\mathsf{Out}(\langle A, B, m\rangle)]$$
$$[\mathsf{In}(\langle A, B, m\rangle)] -[\ \mathsf{Rcv_I}(A, B, m)\ ]\!\rightarrow [\mathsf{Rcv_I}(A, B, m)]$$

Note that the adversary learns the message since we use the Out and In facts. This is intended because we are modeling an insecure channel.

The recipient can now receive the message with the following rule:

$$[\mathsf{Rcv\_I}(\$A, \$B, m)] -[\ ]\!\rightarrow [\mathsf{Conclusion1}(m)]$$

Instead of using In and Out facts directly, protocol rules using insecure channels will now use the Snd_I and Rcv_I facts. Other channels modeling different properties might now use other ways to transmit messages, such as using a Sec fact for secure messages.

2.3.3 Security Properties

Currently, there are three security properties that we require HISP models to fulfill. These are:

- Functionality: There must exist a trace where a message communicated by a sender S is actually learned by a receiver R.
- Confidentiality: Any values declared secret by a party using the Secret action must not be known by the adversary at any time (see the previously noted lemma).
- Authenticity: We require that if a recipient of a message (R) believes it to be authentic (by using the action Authentic($S,$R,m)), then the sender S must have communicated it at some prior point.

We give a more in-depth description and discussion of security properties in Section 4.

3 Extending Channel Properties

3.1 Current Channel Properties

The HISP model described in [5] currently uses channels with the following four properties:

Insecure channels
These channels do not restrict the adversary in any way. The adversary can inject messages into and read messages from this channel, as well as replay messages or otherwise manipulate the messages in the channel. An insecure channel is modeled with the following rules:

$$[\mathsf{Snd_I}(A,B,m)] -[\ \mathsf{Snd_I}(A,B,m)\ ]\rightarrow [\mathsf{Out}(\langle A,B,m\rangle)] \tag{2}$$
$$[\mathsf{In}(\langle A,B,m\rangle)] -[\ \mathsf{Rcv_I}(A,B,m)\ ]\rightarrow [\mathsf{Rcv_I}(A,B,m)] \tag{3}$$

The insecure channel simply sends all information (sender, receiver, and message) to the adversary, who can then manipulate all three arguments as he wishes.

Confidential channels
The adversary is prevented from learning the contents of messages sent through these channels. However, the adversary may still replay, inject, or block messages. Confidential channels are modeled with these rules:

$$[\mathsf{Snd_C}(A,B,m)] -[\ \mathsf{Snd_C}(A,B,m)\ ]\rightarrow [!\mathsf{Conf}(B,m)] \tag{4}$$
$$[!\mathsf{Conf}(B,m), \mathsf{In}(A)] -[\ \mathsf{Rcv_C}(A,B,m)\ ]\rightarrow [\mathsf{Rcv_C}(A,B,m)] \tag{5}$$
$$[\mathsf{In}(\langle A,B,m\rangle)] -[\ ]\rightarrow [\mathsf{Rcv_C}(A,B,m)] \tag{6}$$

The adversary is prevented from gaining knowledge of the message since it is stored in the Conf fact (recall that the adversary can only learn messages contained in an Out fact). Since this fact is only used in Rules 4, 5 and 6, there is also no way for the adversary to learn the contents of the fact via application of other rules. The fact itself is persistent to allow the adversary to replay the message. Due to Rule 5, the adversary is able to modify the sender of a legitimately created message (the receiver cannot be modified since the channel is confidential, and hence the message may only be read by the intended receiver). The ability to inject messages into the channel is given by Rule 6. Additionally, there is no requirement that an applicable rule must be applied at all. Not applying Rule 5 hence models an attack on the availability of the channel.

Authentic Channels
The adversary is prevented from modifying messages sent through these channels as well as their respective sender, but may learn the contents of a message. The authenticity property is modeled by the following rules:

$$[\mathsf{Snd}_\mathsf{A}(A,B,m)] \dashv[\ \mathsf{Snd}_\mathsf{A}(A,B,m)\ ]\mapsto [!\mathsf{Auth}(A,m), \mathsf{Out}(\langle A,B,m\rangle)] \tag{7}$$

$$[!\mathsf{Auth}(A,m), \mathsf{In}(B)] \dashv[\ \mathsf{Rcv}_\mathsf{A}(A,B,m)\ ]\mapsto [\mathsf{Rcv}_\mathsf{A}(A,B,m)] \tag{8}$$

Authenticity is given since receiving a message over an authentic channel requires the existence of the corresponding Auth fact, which cannot be influenced by the adversary. The adversary gains knowledge of the message due to the Out fact, and can choose the recipient of the message due to being able to supply the parameter B in Rule 8. As before, making the fact persistent simulates the adversary's ability to replay a message.

Secure Channels
Secure channels combine the properties of confidential and authentic channels, and are modeled by the following rules:

$$[\mathsf{Snd}_\mathsf{S}(A,B,m)] \dashv[\ \mathsf{Snd}_\mathsf{S}(A,B,m)\ ]\mapsto [!\mathsf{Sec}(A,B,m)] \tag{9}$$

$$[!\mathsf{Sec}(A,B,m)] \dashv[\ \mathsf{Rcv}_\mathsf{S}(A,B,m)\ ]\mapsto [\mathsf{Rcv}_\mathsf{S}(A,B,m)] \tag{10}$$

Since the adversary may only learn information that is contained in an Out fact and pass information to the protocol using an In fact, storing the message in the Sec fact means that the adversary can neither learn m nor create own messages, leaving only the options of replaying or blocking the message (again, the Sec fact is only used in Rules 9 and 10, meaning the adversary cannot gain knowledge of the contents of the fact via other rules).

3.2 Extended Channel Properties

3.2.1 Proposed Channel Properties

We now extend the channels described in Section 3.1 with additional properties (e.g. availability) to more accurately model the capabilities of the adversary. In particular, we want to include channel properties that allow us to describe an adversary with weaker capabilities than the standard Dolev-Yao adversary (since this is oftentimes the case in real-world scenarios). To do this, we must first look at the potential capabilities of the adversary. Recall that the standard Dolev-Yao adversary has the following capabilities:

- Learn the contents of a channel.
- Inject own messages into the channel.
- Replay messages in a channel.
- Delay messages in a channel.
- Deny the availability of a channel (by blocking messages).

We want to create channels that remove certain capabilities from the adversary. The first two capabilities can already be mitigated by the currently available channels, so we focus on the remaining three. Our goal is to create modular channel properties that can be combined to create a specific kind of channel (e.g. a channel that is confidential and replay-free, but not authentic). This allows for more precise modeling of real world channels, since we can now express new properties in our model (e.g. a trusted device communicating a message to a user expressed as a replay-free and available channel).

Replay-Free Channels
These channels ensure that each individual message can only pass through the channel at most once. We achieve this by declaring the facts produced by the channel rules as linear, rather than persistent. This ensures that a message which is sent once is delivered at most once.

Note that we do not define a replay-free *insecure* channel, since we cannot enforce this property on a channel where the adversary can arbitrarily learn and inject messages. We do define the authentic, confidential, and secure replay-free channels.

The following rules show a replay-free authentic channel:

$$[\mathsf{Snd_{AR}}(A,B,m)] \multimap [\ \mathsf{Snd_{AR}}(A,B,m)\ ] \to [\mathsf{Auth_R}(A,m), \mathsf{Out}(\langle A,B,m\rangle)] \tag{11}$$

$$[\mathsf{Auth_R}(A,m), \mathsf{In}(B)] \multimap [\ \mathsf{Rcv_{AR}}(\mathsf{A},\mathsf{B},\mathsf{m})\ ] \to [\mathsf{Rcv_{AR}}(A,B,m)] \tag{12}$$

Similarly, Rules 13, 14 and 15 express a replay-free confidential channel:

$$[\mathsf{Snd_{CR}}(A,B,m)] \multimap [\ \mathsf{Snd_{CR}}(A,B,m)\ ] \to [\mathsf{Conf_R}(B,m)] \tag{13}$$

$$[\mathsf{Conf_R}(B,m), \mathsf{In}(A)] \multimap [\ \mathsf{Rcv_{CR}}(\mathsf{A},\mathsf{B},\mathsf{m})\ ] \to [\mathsf{Rcv_{CR}}(A,B,m)] \tag{14}$$

$$[\mathsf{In}(\langle A,B,m\rangle)] \multimap [\ \ ] \to [\mathsf{Rcv_{CR}}(A,B,m)] \tag{15}$$

Note that we allow the adversary to repeatedly send messages via Rule 15. We simply require that any message that is unknown to the adversary and sent via the $\mathsf{Snd_{CR}}$ fact cannot be replayed.

An important channel that will be used when analysing HISP topologies in regard to injective agreement will be the replay-free secure channel, which is represented by the following rule:

$$[\mathsf{Snd_{SR}}(A,B,m)] \multimap [\ \mathsf{Snd_{SR}}(A,B,m)\ ] \to [\mathsf{Rcv_{SR}}(A,B,m)] \tag{16}$$

Note that since any facts created by the channel are linear for replay-free channels, and since the adversary cannot interact with the message in the channel in any way (except blocking it), we can simply condense the two rules from the secure channel described in Section 3.1 into one rule.

Available Channels
Available Channels ensure that any message sent will also be received. The rules for available

channels are almost equivalent to those of regular authentic/confidential/secure channels, with the exception of additional Available_Send and Available_Receive actions. These actions are however not in the channel rules, but rather in the protocol rules. For example, a secure available channel would have the following rules:

$$[\mathsf{Snd}_{\mathsf{SAv}}(A,B,m)] -[\ \mathsf{Snd}_{\mathsf{SAv}}(A,B,m)\ ]\rightarrow [!\mathsf{Sec}_{\mathsf{Av}}(A,B,m)] \tag{17}$$

$$[!\mathsf{Sec}_{\mathsf{Av}}(A,B,m)] -[\ \mathsf{Rcv}_{\mathsf{SAv}}(A,B,m)\ ]\rightarrow [\mathsf{Rcv}_{\mathsf{SAv}}(A,B,m)] \tag{18}$$

A Send rule using this secure available channel would look like this:

$$[\mathsf{AgentState}(\$S,\ldots),\ldots] -[\ \mathsf{Available\_Send}(A,B,m),\ldots\ ]\rightarrow [\mathsf{Snd}_{\mathsf{SAv}}(A,B,m),\ldots]$$

Simply adding actions will however not ensure availability. To enforce that a sent message is also received, we need to add the following axiom:

```
axiom available:
"All A B m #i.  Available_Send(A,B,m) @i ==> Ex #j.  Available_Receive(A,B,m) @j & i<j"
```

This axiom ensures that there must be a Receive action for every Send action, and hence guarantees that the channel is available.

Immediate Channels

These channels ensure that the adversary cannot hold back individual messages in the channel. More precisely stated, it should not be possible to execute any protocol rule between the sending and receiving of a message in an immediate channel. This is achieved by first adding a Tic action to every protocol rule. This serves to differentiate protocol rules from any other rules so that every execution of a protocol rule can be seen in the trace. While it would be sufficient to simply define the action without parameters, we add the agent executing the protocol rule as an argument to the Tic action (this will later help us define certain security properties).

Additionaly, we again move the Send and Receive actions out of the channel rules and into the protocol rules. To illustrate, the following rules formalize the authentic immediate channel:

$$[\mathsf{Snd}_{\mathsf{AI}}(A,B,m)] -[\ \ ]\rightarrow [\mathsf{Auth}_{\mathsf{I}}(A,m), \mathsf{Out}(\langle A,B,m\rangle)] \tag{19}$$

$$[\mathsf{Auth}_{\mathsf{I}}(A,m), \mathsf{In}(B)] -[\ \ ]\rightarrow [\mathsf{Rcv}_{\mathsf{AI}}(A,B,m)] \tag{20}$$

A Send rule using immediate channels would look like this:

$$[\mathsf{AgentState}(\$S,\ldots),\ldots] -[\ \mathsf{Tic}(\$S), \mathsf{Immediate\_Send}(A,B,m),\ldots\ ]\rightarrow [\mathsf{Snd}_{\mathsf{AI}}(A,B,m),\ldots]$$

Now that we have all protocol rules marked accordingly, we can create an axiom to guarantee that no protocol rule may be executed between an immediate send and its corresponding immediate receive:

```
axiom immediate:
"All A B m #i #j #k.  Immediate_Send(A,B,m) @i & Immediate_Receive(A,B,m) @j & i<j
& Tic() @k & i<k & k<j ==> F"
```

This axiom states that there may be no Tic action which is strictly in between an Immediate_Send and an Immediate_Receive action, and thus, no protocol rule is allowed to execute during a use of an immediate channel. In real-world terms, this corresponds to the adversary not being able to hold back the information gained from the channel and use it somewhere else, before it has reached its intended destination.

3.2.2 Justifications

In this section, we justify our choice of the proposed channel properties by arguing that it is not possible to express one property with the other two properties. We show this by proposing real-life scenarios showcasing the differences between the properties.

Replay-Free Channels
To show that immediate, available channels cannot replace replay-free channels, consider a regular bank transaction over a secure, but replayable channel. H sends a command to transfer money to some untrustworthy business. The adversary (who may be working for the business) can intercept and replay the message to get more money. Guaranteeing that the channel is immediate and available does not prevent this attack, since the adversary executes the attack after the regular protocol has completed normally.

Immediate Channels
Let us regard a real-world example found in multiplayer videogames. The adversary (a dishonest player), can delay packets containing game data from and to his device using a "lagswitch", which temporarily disables the connection. This results in all other players becoming immobile (since no new packets arrive at the device), allowing the adversary to defeat the other players with ease locally. Then, the connection is re-established and the data from the adversary (i.e. that he defeated the other players) is transmitted to the server. Obviously, this attack could be mitigated by using an immediate channel.

We see that using an available, replay-free channel does not mitigate this attack, since messages are delayed, but do arrive and are sent only once.

Available Channels
We show that we cannot achieve the functionality of an available channel using immediate, replay-free channels. Assume we have a car door that is locked by a "secure" pulse from a car key. H presses the "lock"-button on the key and then leaves. However, if the adversary has jammed the signal, it will die without ever reaching the car. The adversary can now simply steal the contents of the open car. Using immediate, replay-free channels will not mitigate this attack, since the only requirements for these channels are that:

1. No step in the protocol is executed until the immediate message has arrived at its recipient. However, the message (i.e. the lock signal) never actually has to arrive at the car.
2. A unique message is never sent more than once. However, no message is sent by the adversary at all.

4 Extending Security Properties

In the previous section, we described channels that have a certain property by definition, e.g. being confidential or secure. Now, we will regard these properties as goals that a protocol must fulfill. We formally define a security property with two functions p and q, as introduced in [5]. Function p denotes an event that must occur in at least one trace of a protocol, while function q denotes

an event that must occur in every trace. More formally, a protocol $\mathcal{R}$ provides a channel with the property defined by (p, q) if

$$\exists\, tr \in TR(\mathcal{R}), S, R \in \mathit{Honest}(tr), m \in \mathcal{M}\colon p(tr, S, R, m) \wedge$$
$$\forall\, tr \in TR(\mathcal{R}), S, R \in \mathit{Honest}(tr), m \in \mathcal{M}\colon q(tr, S, R, m).$$

The following overview will only contain the formal description of the current security properties for brevity. The actual implementation of these properties in Tamarin models can be found in [5].

4.1 Existing HISP Security Properties

Currently, the HISP model contains four basic, composable security properties. The channels fulfilling these properties are:

Communication Channels
This functionality property expresses that channels are capable of communicating a message from a sender to a recipient. In particular, this means that the sender must have sent the message before the recipient learns the message. In the notation used in [5], we can define communication channels by the property $(p_{\text{comm}}, q_{\text{comm}})$, where

$$p_{\text{comm}}(tr, S, R, m) := \mathsf{Communicate}(tr, S, R, m)$$
$$q_{\text{comm}}(tr, S, R, m) := \top$$

Originating Channels
The originating channel property allows the sender to generate a fresh value, symbolically representing the ability to generate an arbitrary message. Note that while this property is often implicitly contained in other models and protocols, this is not the case for all HISP topologies, since some of them use a "code-voting" approach where messages are predefined. This makes sense in a voting environment because often the parameters (e.g. candidate names, or a simple "yes" or "no") are known beforehand. Originating channels are defined by the property $(p_{\text{orig}}, q_{\text{orig}})$, where

$$p_{\text{orig}}(tr, S, R, m) := \text{fresh}(S, m) \in tr$$
$$q_{\text{orig}}(tr, S, R, m) := \top$$

Confidential Channels
Confidentiality is a safety property, and states that an adversary may not learn any message transmitted through a channel with this property. Messages intended to remain secret are declared by the action $\mathsf{Secret}(S, R, m)$. The property itself is defined as $(p_{\text{conf}}, q_{\text{conf}})$, where

$$p_{\text{conf}}(tr, S, R, m) := \mathsf{Secret}(S, R, m) \in tr$$
$$q_{\text{conf}}(tr, S, R, m) := \mathsf{Secret}(S, R, m) \in tr \rightarrow \mathsf{!K}(m) \notin tr.$$

Authentic Channels
The authenticity property described in [5] states that if a recipient of a message believes it to be authentic, then the sender must have communicated it at some prior point. Formally, the property is defined as $(p_{\text{auth}}, q_{\text{auth}})$, where

$$p_{\text{auth}}(tr, S, R, m) := \mathsf{Authentic}(S, R, m) \in tr$$
$$q_{\text{auth}}(tr, S, R, m) := \mathsf{Authentic}(S, R, m) \in tr \rightarrow \mathsf{Communicate}(tr, S, R, m) \in tr.$$

As before, messages intended to be authentic are declared by the action $\mathsf{Authentic}(S, R, m)$.

4.2 Extended Security Properties

One of the goals of this thesis is to expand on the previously shown notion of authenticity, since the current definition is insufficient to express concepts like *injectivity* or *agreement*. In the following, we formalize various levels of authentication in the HISP model.

4.2.1 General Model Extensions

In order to implement some of the new security properties, a few additions to the HISP model had to be made.

Binding Agents to Roles
For some of the following properties, we must determine if an agent is in a specific role. We achieve this by explicitly pairing an agent with a role in an action. A role is defined in the model as an arbitrary public value, e.g. $'Role1'$ or $'Sender'$. The binding of an agent to a role occurs with an action $\mathsf{Role}(\$Agent, 'role')$ in the setup rule.

Another extension we make is to add thread IDs (or tids) to each run of a protocol. Since Tamarin may create multiple simultaneous runs of the same protocol, tids allow us to distinguish each individual run rather than just being able to distinguish by roles. A tid is created during the setup rule and is stored in the agent state for the entire duration of the protocol, e.g. as AgentState($S,'S_0','empty',tid_S). Using a tid allows us to create security properties dealing with *injectivity* and *synchronization*, such as *injective agreement*. For the recentness properties, we will additionally redefine the Tic action to include the tid as a parameter, e.g. as Tic($S,tid_S).

4.2.2 Weak Authentication Properties

While the ultimate goal of this section is to find authentication properties that are stronger than the one described in [5], we have also decided to create Tamarin lemmas for the weaker properties proposed by Lowe in [4]. These properties may help answer some interesting questions about HISP topologies which are outside the scope of this thesis, such as if we can further minimize the topologies in [5] if we only require them to fulfill weak aliveness.

Note that while the weak aliveness properties are weaker than the authenticity property described in [5], the recentness properties are not directly comparable to this property.

Also note that for all of the following properties, the Tamarin lemmas only model the q function of the formal definition. The p function is modeled by a separate lemma in all protocols.

Weak Aliveness
If an agent S claims weak aliveness with some other agent R (in Tamarin, via an action Weak_Alive(S,R)), then at some point in time, R must have performed some action. This property can be formally defined as $(p_{\text{wAlive}}, q_{\text{wAlive}})$, where

$$p_{\text{wAlive}}(tr, S, R, m) := \mathsf{Weak\_Alive}(S, R) \in tr$$
$$q_{\text{wAlive}}(tr, S, R, m) := \mathsf{Weak\_Alive}(S, R) \in tr \rightarrow \mathsf{Tic}(R) \in tr.$$

In Tamarin, this property can be described with the following lemma:

```
lemma weak_alive:
"All S R #k.  Weak_Alive(S,R) @k ==> (Ex #i.  Tic(R) @i)"
```

Weak Aliveness in Correct Role
In this scenario, an agent S claims weak aliveness in correct role with another agent R in role $'Role'$. We express this in Tamarin with the action Weak_Alive_Role(S,R,'Role'). Similarly to the previous scenario, an agent R must have performed some action at some point in time, but this time in role *Role* (i.e. there must be an action Role(R,'Role') in the trace). This property can be formally defined as $(p_{\text{wAliveRole}}, q_{\text{wAliveRole}})$, where

$p_{\text{wAliveRole}}(tr, S, R, m) := \exists role : \mathsf{Weak\_Alive\_Role}(S, R, role) \in tr$
$q_{\text{wAliveRole}}(tr, S, R, m) := \forall role : \mathsf{Weak\_Alive\_Role}(S, R, role) \in tr \rightarrow \mathsf{Tic}(R) \in tr \wedge \mathsf{Role}(R, role) \in tr.$

In Tamarin, we describe this property with the lemma:

```
lemma weak_alive_role:
"All S R role #k.  Weak_Alive_Role(S,R,role) @k ==> (Ex #i #j.  Tic(R) @i & Role(R,role)
@j)"
```

Recent Aliveness
If an agent S claims recent aliveness with an agent R, then R must have performed at least one action, namely before the claim was made, but after another action performed by S in the same trace. This property can be formally defined as $(p_{\text{rAlive}}, q_{\text{rAlive}})$, where

$p_{\text{rAlive}}(tr, S, R, m) := \exists tid\_S : \mathsf{Recent\_Alive}(S, R, tid\_S) \in tr$
$q_{\text{rAlive}}(tr, S, R, m) := \forall tid\_S : \mathsf{Recent\_Alive}(S, R, tid\_S) \in tr \rightarrow \exists tr', tr'', tr''', tid\_R : tr = tr' \cdot tr'' \cdot tr'''$
$\wedge \mathsf{Tic}(S, tid\_S) \in tr' \wedge \mathsf{Tic}(R, tid\_R) \in tr'' \wedge \mathsf{Recent\_Alive}(S, R, tid\_S) \in tr'''.$

In Tamarin, this property can be described with the following lemma:

```
lemma recent_alive:
"All S R tid_S #k.  Recent_Alive(S,R,tid_S) @k ==> (Ex #i #j tid_R. Tic(S,tid_S) @i
& Tic(R,tid_R) @j & i<j & j<k)"
```

Recent Aliveness in Correct Role
If an agent S claims recent aliveness with another agent R in role $'role'$, then as before, R must have performed an action before the claim but after another action of S. Additionally, R must have been in role $'role'$. This property can be formally defined as $(p_{\text{rAliveRole}}, q_{\text{rAliveRole}})$, where

$p_{\text{rAliveRole}}(tr, S, R, m) := \exists role, tid\_S : \mathsf{Recent\_Alive\_Role}(S, R, tid\_S, role) \in tr$
$q_{\text{rAliveRole}}(tr, S, R, m) := \forall role, tid\_S : \mathsf{Recent\_Alive\_Role}(S, R, tid\_S, role) \in tr \rightarrow \exists tr', tr'', tr''', tid\_R :$
$tr = tr' \cdot tr'' \cdot tr''' \wedge \mathsf{Tic}(S, tid\_S) \in tr' \wedge \mathsf{Tic}(R, tid\_R) \in tr'' \wedge \mathsf{Recent\_Alive\_Role}(S, R, tid\_S, role) \in tr'''$
$\wedge \mathsf{Role}(R, role) \in tr.$

In Tamarin, this property can be described by the following lemma:

```
lemma recent_alive_role:
"All S R role #k.  Recent_Alive(S,R,tid_S,role) @k ==> (Ex #i #j #l tid_R. Tic(S,tid_S)
@i & Tic(R,tid_R) @j & i<j & j<k &Role(R,role) @j)"
```

4.2.3 Strong Authentication Properties

One of the goals of this thesis was to create stronger properties for authentication. We therefore propose models for non-injective agreement and synchronization, as well as injective agreement and synchronization. All following security properties will use Commit and Running actions to establish agreement on certain information. The syntax for these actions is: $\mathsf{Commit}(S, R, data, tid\_S)$ and $\mathsf{Running}(S, R, data, tid\_R)$.

For these actions, we use the previously introduced tids, which allows us to pair individual runs together for the injective properties. From a semantic viewpoint, these actions express the commitment to certain data (i.e. S committing via this action means that he believes to share the same data with R). The Running action denotes that R actually believes he has provided the data to a committing party S. Using these actions, we can determine an attack on the authenticity property if e.g. there is a Commit action without a corresponding Running action (which implies that R never actually sent the message that S has committed to). For a more in-depth explanation of this mechanism, we refer to [4].

Non-Injective Agreement
Informally, non-injective agreement (or NIA for short) requires that if an Agent A in role S has run a protocol apparently with Agent B, then Agent B has also run the protocol (in role R), and A and B agree on the data exchanged in that protocol run.

Expressed in actions, this means that for every Commit action between two parties on some data, there must have been a prior Running action between the same parties containing the same data. No further restrictions are imposed. In particular, replaying a message would not constitute an attack against this property as long as all Commit actions occur after the Running action. Formally, this definition corresponds to $(p_{\text{NIA}}, q_{\text{NIA}})$, where

$$p_{\text{NIA}}(tr, S, R, m) := \exists tid\_S : \mathsf{Commit}(S, R, m, tid\_S) \in tr$$
$$q_{\text{NIA}}(tr, S, R, m) := \forall tid\_S : \mathsf{Commit}(S, R, m, tid\_S) \in tr \rightarrow \exists tr', tr'', tid\_R : tr = tr' \cdot tr'' \wedge \mathsf{Running}(S, R, m, tid\_R) \in tr' \wedge \mathsf{Commit}(S, R, m, tid\_S) \in tr''.$$

In Tamarin, we can express this property with the following lemma:

```
lemma noninjective_agree:
"All S R data tid_S #k.  Commit(S,R,data,tid_S) @k ==> (Ex #i tid_R.
Running(S,R,data,tid_R) @i & i<k)"
```

Injective Agreement
With injective agreement, we informally want to express that a protocol is resistant to replay attacks. Using the previous definition of NIA, we additionally require that every run of Agent A corresponds to a unique run of Agent B. Therefore, we require that every thread issuing a Commit action has a corresponding thread issuing a Running action (as opposed to NIA, where the only requirement was that there had to be at least one prior Running action). More formally, this means

that we require the existence of an injective function from the space of all Commit actions to the space of all Running actions.

Before we define injective agreement, we need to allow the partitioning of traces into smaller subtraces:

Definition 2. *Let tr be a trace that has been arbitrarily partitioned into n parts. We denote tr_i as the i-th partition and $tr_{\{i...j\}}$ as the concatenation of all partions between and including tr_i and tr_j.*

We can now define injective agreement with $(p_{\text{IA}}, q_{\text{IA}})$ as follows:

$$p_{\text{IA}}(tr, S, R, m) := \exists tid\_S : \mathsf{Commit}(S, R, m, tid\_S) \in tr$$
$$q_{\text{IA}}(tr, S, R, m) := \forall tS : \mathsf{injective}(tr, S, R, m, tS)$$

where

$$\mathsf{injective}(tr, S, R, m, tS) := \exists f_{tr}\{i | \mathsf{Commit}(S, R, m, tS) \in tr_i\} \rightarrow$$
$$\{j : \mathsf{Running}(S, R, m, tD) \in tr_j\}\mathsf{injective}$$

The injective agreement property used in our protocol models corresponds very closely to this definition, with the added requirement that each paired Running action must occur before the Commit action it is paired with. This change is reflected in the following modification of the injective function:

$$\mathsf{injective}(tr, S, R, m, tS) := \exists f_{tr}\{i | \mathsf{Commit}(S, R, m, tS) \in tr_i\} \rightarrow$$
$$\{j : \mathsf{Running}(S, R, m, tD) \in tr_j\}\mathsf{injective} \wedge \forall i : \mathsf{Commit}(S, R, m, tS) \in tr_i \rightarrow$$
$$\exists \mathsf{Running}(S, R, m, tD) \in tr_{f(i)} : i > f(i)$$

These two formulas are logically equivalent provided there are no trace restrictions that require the existence of future actions (e.g. the availability property, which always requires a message to be received after it has been sent).

Proof. Let tr be an arbitrary, fixed trace. Assume that there are no restrictions as described above on tr. We make the following case distinction:

- Let there be S,R,m s.t. there are more Commit actions on S,R,m than Running actions. Since f_{tr} must be injective by definition, it is obvious that no such function can exist for this trace (the domain is larger than the image of the function). Hence, both formulas will return $false$.
- Let there be at least as many Running actions as there are Commit actions for all S,R,m. We can now do another case distinction:
  a. For every Commit action, there is a previously occurring matching Running action. This means that the function f_{tr} exists, and since we only regard commits where the Running action occurs beforehand, we automatically have both requirements for injectivity fulfilled. Since all Commit actions have their Running action beforehand, it is not possible to truncate the trace in such a way that we have a Commit action without a corresponding Running action. In this case, both formulas will return $true$.
  b. There exists at least one Commit action s.t. its corresponding Running action occurs afterward. For simplicity, let us assume that this is the last Commit action in the trace (if this is not the case, we can define a truncated trace tr'' where all further commits

have been removed). Let j be the index where the first corresponding Running action occurs. We now define a trace $tr' := tr_{\{1...(j-1)\}}$. This is a valid trace, and it has been created in such a fashion that there exists a Commit action without a corresponding Running action. Since the injectivity function must hold for all traces, we know that the protocol cannot be injective. Again, both formulas will return $false$.

Since both formulas return the same values for all possible cases, we have shown that they are logically equivalent. □

Now we can describe the implementation of injective agreement as a security property. Using the previous definition, we know that:

a. For each Commit action, there must be a unique Running action. A trace where more Commit actions than Running actions occur is not injective.

b. No two Commit actions may be paired with the same Running action.

c. For each Commit action, the Running action it is paired to must occur beforehand.

d. All paired Running and Commit actions must have the same parameters.

First, note that we disallow multiple identical Commit actions (i.e. actions that match in all parameters) in a run, since it would be impossible to distinguish these Commit actions. In general, this is not a problem since each party usually commits at most once. Our goal is for the injective agreement lemma to check whether all commits on a certain piece of data have a corresponding Running action, and additionally check that each commit is paired to a Running action with the same data.

Since all these pairs are unique by definition, each Commit action will correspond to a unique Running action. The lemma fails if there is a commit that has no corresponding Running action (agreement), or if there is no pairing between the Commit and any suitable Running action (injectivity).

The pairing of Running and Commit actions is recorded by an action $\mathsf{Pair}(tid\_S, tid\_R)$. Since we want Tamarin to be able to pair arbitrary runs, we do the pairing via a protocol rule. First, we must manually add Cmt and Rng facts to the protocol rules containing the Commit and Running actions. This rule will contain the same information as the corresponding action, with an additional identifier for the security property we are checking. For example, a committing rule may look as follows:

$$[\mathsf{AgentState}(\$H, 'H\_3', data, tid\_H), \ldots] -[\ \mathsf{Commit}(\$H, \$S, data, tid\_H)\ ]\rightarrow$$
$$[\mathsf{Cmt}(\$H, \$S, tid\_H, 'IA', data), \ldots]$$

Note that the fourth parameter in the Cmt fact denotes the type of agreement that is being committed to. While not strictly necessary, this allows us to have different types of commits using the same parameters. The pairing itself now occurs in a separate rule, shown below:

$$[\mathsf{Cmt}(S, R, tid\_S, type, data), \mathsf{Rng}(S, R, tid\_R, type, data)] -[\ ]\rightarrow$$
$$[\mathsf{Pair}(tid\_S, tid\_R), \mathsf{PairedRunning}(S, R, data, tid\_R)]$$

Now, we can verify that every Commit action is indeed paired with a Running action with the lemma:

```
lemma injective_agree:
"All S R data tid_S #k.  Commit(S,R,data,tid_S) @k ==> (Ex tid_R tid #i #j.
Running(S,R,data,tid_R) @i & i<k & Pair(tid_S,tid) @j)"
```

Observe that the tid for the recipient used in the Pair action does not have to correspond to the tid in the Running action. This prevents "false negatives" due to incorrect pairing. To illustrate, assume there are Commit actions by two different runs on the same data at $t = 2$ and $t = 4$, and corresponding Running actions at $t = 1$ and $t = 3$. Since Tamarin can pair these runs arbitrarily, it might pair the runs as $(C@2, R@3), (C@4, R@1)$, and enforcing the same tid for the recipient in the lemma would cause Tamarin to report an attack (since one Commit occurs before its paired Running), even though this trace satisfies injective agreement.

Another problem is the fact that Tamarin can simply refuse to pair two runs, as it can choose to not execute the pairing rule. This is obviously an undesireable state that can lead to incorrect results, so we fix this by adding the following axiom:

```
axiom force_pairing:
"(All S R data tid_S #k.  Commit(S,R,data,tid_S) @k ==> (Ex tid #i.  Pair(tid_S,tid)
@i) | (All tid_R #j.  Running(S,R,data,tid_R) @j ==> (Ex #l.
PairedRunning(S,R,data,tid_R) @l )))"
```

This axiom states that for any Commit action, the *tid_S* within it must either be paired, or all tids used in matching Running actions must already be used up in a pair (denoted by the PairedRunning action). Any trace with unpaired matching runs is discarded.

Combining all these elements allows us to test protocols for injective agreement. Examples of such protocols can be found at [6].

Synchronization

While our main focus for authentication was on agreement, we also tried to create synchronization properties. Informally, the only difference to agreement is that all messages between two runs must be sent and received in the correct order, i.e. it should not be possible that a party receives a message before the sender has sent it.

In Tamarin, we must first give ourselves the ability to track the sending and receiving of messages for each individual thread. This can be done by simply adding the tid to the Snd and Rcv facts as a fourth parameter. Additionally, we only need one lemma for synchronization if we add generic Send and Recv actions to the protocol rules where Snd and Rcv facts are used, in the same manner as the actions for injective agreement. Otherwise, we would need a synchronization lemma for every channel defined in the protocol.

The only additional guarantee we need to make for synchronization is that all Snd and Rcv facts of any paired runs occur in the right order. This means we can use the same setup for injective and non-injective synchronization as for agreement, with some minor changes.

For non-injective synchronization we use the following lemma:

```
lemma noninjective_synch:
"All S R data tid_S #k.  Commit(S,R,data,tid_S) @k ==> (Ex #i tid_R.
Running(S,R,data,tid_R) @i & i<k
```

```
& (All m #l.  Recv(S,R,m,tid_R) @l ==> Ex #j.  Send(S,R,m,tid_S) @j & j<l)
& (All m #l.  Recv(R,S,m,tid_S) @l ==> Ex #j.  Send(R,S,m,tid_R) @j & j<l))"
```

Similarly, for injective synchronization we can use the following lemma:

```
lemma injective_synch:
"All S R data tid_S #k.  Commit(S,R,data,tid_S) @k ==> (Ex tid_R tid #i #j.
Running(S,R,data,tid_R) @i & i<k & Pair(tid_S,tid) @j
& (All m #l.  Recv(S,R,m,tid_R) @l ==> Ex #j.  Send(S,R,m,tid_S) @j & j<l)
& (All m #l.  Recv(R,S,m,tid_S) @l ==> Ex #j.  Send(R,S,m,tid_R) @j & j<l))"
```

5 Characterization of Topologies

5.1 Characterization with Respect to Security Properties

In this section, we subject all HISP protocol models of [5] to our formalization of non-injective agreement and injective agreement given in Section 4.2.3, and fix protocols that do not fulfill either of the properties while retaining their underlying topology.

5.1.1 Results for Unmodified Protocols

For this verification step, all protocols described in [5] were extended with the previously created agreement rules and lemmas. Originally, we tried using Tamarin's automated verification, with all tests running for a fixed amount of time before being aborted, or until Tamarin had run out of available memory. However, since many protocols failed to verify within the time limit, we eventually proved all protocols manually. All models with their corresponding proofs can be found at [6]. Table 1 shows an overview of the results.

5.1.2 Ensuring Non-injective Agreement

All protocols that did not satisfy NIA were suffering from the same problem, namely that the party issuing the Running action wishes to communicate with a party different from the one issuing the Commit action, resulting in mismatching Running and Commit actions. For example, H might want to communicate with S, and issues an action $\mathsf{Running}(S, H, tid\_H, m)$. In this scenario, D (which is used by H) is actually used to communicate with another server $S.1$, which produces the mismatching commit fact $\mathsf{Commit}(S.1, H, tid\_S.1, m)$. In real-life terms, this would correspond to H wishing to speak to server 1, but using a device which is part of the infrastructure of server 2. This might happen by mistake, e.g. if the human uses multiple E-Banking services, or intentionally, if an attacker can somehow exchange the correct device for a corrupted one, linked to a phishing site (the protocol would still be technically safe, only that the connection is now safely established to a server owned by the adversary).

Protocol Name	Result for NIA	Result for IA
Lemma 4a	verified	falsified
Lemma 4b	verified	falsified
Lemma 4c	verified	verified
Lemma 5a	verified	falsified
Lemma 5b	verified	falsified
Lemma 5c	verified	falsified
Lemma 6a	verified	verified
Lemma 6b	verified	falsified
Lemma 7a	verified	falsified
Lemma 7b	verified	falsified
Lemma 7c	falsified	falsified
Lemma 8a	verified	verified
Lemma 8b	verified	verified
Lemma 8c	verified	verified
Lemma 8d	verified	verified
Lemma Da	verified	falsified
Lemma Db	verified	falsified
Lemma Ea	falsified	falsified
Lemma Eb	verified	falsified
Transauth a	falsified	falsified
Transauth b	falsified	falsified
CodeVoting	verified	verified
Example 2	falsified	falsified
Example 3	verified	verified

Table 1: Results for all protocols described in [5].

The solution to this problem is to have the trusted device explicitly state the intended communication partner S beforehand. For example, in Lemma *Ea*, we would change rule *D2* from:

$$[\mathsf{AgentState}(\$D, 'D\_1', \langle \$H, \$S, k, m\rangle, tid\_D)] -[\ ]\rightarrow$$
$$[\mathsf{Snd\_S}(\$D, \$H, \langle m, h(k, m, \$S, \$D, \$H)\rangle), \mathsf{AgentState}(\$D, 'D\_2', \langle \$H, \$S, k, m\rangle, tid\_D)]$$

to the following:

$$[\mathsf{AgentState}(\$D, 'D\_1', \langle \$H, \$S, k, m\rangle, tid\_D)] -[\ ]\rightarrow$$
$$[\mathsf{Snd\_S}(\$D, \$H, \langle m, h(k, m, \$S, \$D, \$H)\mathbf{,\$S}\rangle), \mathsf{AgentState}(\$D, 'D\_2', \langle \$H, \$S, k, m\rangle, tid\_D)]$$

This way, the human recipient of the message can check the server specified in the message against the intended server, and will not proceed if they do not match. Implementing this modification resulted in non-injective agreement being satisfied for all HISP topologies. The protocols used can be found at [6].

5.1.3 Ensuring Injective Agreement

We next tested all modified protocols from Section 5.1.2 for injective agreement. These tests were all proved manually due to Tamarin's automated verification not terminating. We found that in

addition to the protocols indicated as "verified" in Table 1, only the modified version of protocol *Example 2* also fulfilled injective agreement. Note that the verification of protocol *Lemma 7b* did not terminate.

Our next goal was to repair all protocols which did not fulfill injective agreement while still maintaining the underlying topology (i.e. only the channels defined in the protocol's respective topology could be used). We restricted ourselves to protocols providing non-originating channels. Such channels suffice for entity authentication, where the message that is communicated (i.e. the identity of the sender) need not be fresh.

While all protocols have different underlying topologies, we found that there was only a small number of methods required to ensure injective agreement for all protocols. In the following, we list all the used methods:

1. Declaring Channels between H and D Replay-free
While in general we aimed to create replay-free channels by modifying the protocol itself, in this case we had to explicitly change the topology by defining channels between H and D as replay-free for some protocols. This was done with the previously defined channel property 16. The reason for this is that H must have no initial state, and thus cannot use mechanisms like counters, which require shared state.

2. Adding Counters to Channels between D and S
In order to make channels between D and S replay-free without requiring multiple messages to pass between them, we use the concept of a previously agreed on counter value (this will be described in detail in Section 5.1.5). For all protocols we model these counters with a shared nonce.

3. Agreement on D
This change is necessary for some protocols because the current implementation of injective agreement allows for an incorrect pairing of Commit and Running actions, resulting in "false positives" for a protocol that is actually secure. The issue lies with the fact that we use a modified IA lemma, as shown below:

```
lemma injective_agree:
"All S R data tid_S #k.  Commit(S,R,data,tid_S) @k ==> (Ex tid_R tid #i #j.
Running(S,R,data,tid_R) @i & i<k & Pair(tid_S,tid) @j) | (Ex #l.  Dishonest(S) @l)
| (Ex #l.  Dishonest(R) @l) | (Ex T #l.  Trust(T) @k & Dishonest(T) @l)"
```

This lemma is modified to allow a Commit action to remain unpaired if the committing party (here H) trusts an agent that has become dishonest. Since there is no restriction on how Tamarin pairs matching Commit and Running actions, it can be that it pairs a Running action with a Commit action caused by a party using a compromised device. The underlying issue is that we do not include enough information to make the Commit/Running pairs unique, allowing them to be mapped to several different runs and thus losing the injectivity property.

To illustrate, consider the following scenario: There are two devices, a human and a server. S sends a message to H via the secure device. However, this message is copied and also sent to H by a compromised device. H will now commit to both received messages, even though it was sent only once. The way the lemmas and axioms for injective agreement are written does not prohibit

Tamarin from matching the commit caused by the corrupted device to the Running action issued by S. However, the original, clean commit now no longer has a corresponding Running, and the IA property is falsified. Obviously, the "correct" pairing in this scenario would be the Running action together with the Commit action coming from the uncompromised device, and leaving the other Commit action unpaired, resulting in a verification of the IA property.

The method for fixing this flaw involves adding the device D to the Commit and Running rules and actions. For example, a Commit action by H might now look as follows: $\mathsf{Commit}(\$H, \$S, \langle m, \$D\rangle, tid\_H)$.

This results in the correct pairing of the messages sent and received by the honest device. In real-world terms, this change means that both human and server are somewhat aware of where a message is "supposed to come from", i.e. what device is supposed to be used.

5.1.4 Results for Injective Agreement

Using various combinations of the methods described, we can guarantee injective agreement for all of the modified protocols from Section 5.1.2 except protocol *Lemma 7b*. The various protocols and the methods used to ensure injective agreement are listed below in Table 2. Note that protocol *Example 2* fulfilled IA directly after fixing the issue with NIA, and protocol *Lemma 7b* did not terminate.

Protocol Name	**IA Fulfilled**	**Modifications**
Lemma 4a	Yes	1 and 2
Lemma 4b	Yes	1
Lemma 5a	Yes	1
Lemma 5b	Yes	1
Lemma 5c	Yes	1 and 2
Lemma 6b	Yes	2
Lemma 7a	Yes	3
Lemma 7b	Unknown	-
Lemma 7c	Yes	2 and 3
Lemma Da	Yes	1 and 2
Lemma Db	Yes	1
Lemma Ea	Yes	3
Lemma Eb	Yes	3
Transauth b	Yes	3
Example 2	Yes	-

Table 2: IA results for modified protocols.

5.1.5 Counter Concepts

Since the definition of injective agreement requires that each run of a sender must correspond to a run of the receiver, we must prevent messages from being replayed. One way of preventing replay is to extend the sent messages with counters. The idea is simple: If the sender has a value that always increases when it is transmitted, and the receiver checks and updates his corresponding value, then it should never be possible for the receiver to accept the same message twice (since the value of the

message will be lower than the expected value). We present two possible models of such a counter:

Multisets
In this approach, we use the multiset functionality provided by Tamarin. The number of elements in the multiset will represent the current value of the counter. Since we can compare the sizes of different multisets, we can reason about whether or not a message has a sufficient counter value and should be accepted.

A counter should be shared between any two parties A and B. It should be bi-directional and contain a strictly increasing value. However, both A and B should store the value separately. Based on these requirements, we propose the following facts as a counter between A and B:

$$\mathsf{Counter}(\$A,\$B,'x'),\ \mathsf{Counter}(\$B,\$A,'x')$$

This is a counter for a shared channel between A and B. The first fact models the value that A believes the counter to have, currently the value '1'. Similarly the second fact models what B believes the shared counter's value is. With this design, the counter is bi-directional and per-channel, but its value is stored separately by A and B, and the stored values may differ.

Using this counter model, we first design the counter creation rule. It looks as follows:

$$[\,]\,{-[}\ \mathsf{CC}(\$A,\$B)\ {]\!\rightarrow}\,[\mathsf{Counter}(\$A,\$B,'x')]$$

This rule creates the initial counter value belonging to A (for a channel between A and B). To ensure that only one such counter is created per owner and channel, we add the following axiom to the model:

```
axiom one_counter:
"All A B #i #j.  CC(A,B) @i & CC(A,B) @j ==> #i=#j"
```

In order for a message to be accepted by a recipient, we require that the counter value in the sent message is strictly larger than the value stored by the recipient. We fulfill this requirement in the protocol rules by adding a constant to the multiset of the sender (and thereby increasing the value of the counter) before sending the message. This is illustrated by the following rule:

$$[\mathsf{Counter}(\$A,\$B,count),\ldots]\,{-[}\ {]\!\rightarrow}\,[\mathsf{Snd\_S}(\$A,\$B,\langle m, count+'x'\rangle),\mathsf{Counter}(\$A,\$B,count+'x')]$$

Note that we are increasing the counter value by adding the public value 'x' to the multiset using the "+" operator.

The recipient's rule looks very similar. The difference is that the counter value of the message is accepted as the new counter value, instead of manually updating the value by adding a constant to the multiset. This is done to keep the functionality of the counter intact even if manipulations of the channel occur. Consider for example the adversary deleting a message from the channel. The next sent message would have a counter value of +2 compared to the recipient. We want the recipient to accept the message (since the value is larger than its own stored value), but simply increasing the value manually would allow the attacker to replay the message (since the value in the message would then still be larger). The recipient's rule looks as follows:

$$[\mathsf{Rcv\_S}(\$A,\$B,\langle m, xcount\rangle),\mathsf{Counter}(\$B,\$A,count)]\,{-[}\ \mathsf{Compare}(xcount,count)\ {]\!\rightarrow}$$
$$[\mathsf{Counter}(\$B,\$A,xcount)]$$

Since we assume that the recipient has some internal method of determining whether or not a counter value is valid, we can model this requirement with an axiom:

```
axiom verify_counter:
"All nC cC #i.  (Compare(nC,cC) @i ==> (Ex x.  cC+x=nC))
```

This axiom ensures that any trace where a received message has a counter value smaller or equal to the one stored by the recipient is discarded.

Unfortunately, using this variant in the models causes Tamarin to enter an endless loop, and thus yields no results. The reason for this is currently unknown. Our solution was to model the counter with nonces, described below.

Nonces
The alternate and more simplistic approach to modeling counters is simply to use shared nonces. A message will then only be accepted by the recipient if his stored nonce matches the one sent in the message. This variant is stricter than a realistic counter model, since we cannot express that one value must be larger than the other, the values must match exactly. We use this approach to prove the existence of protocols providing injective agreement.

Note that this solution is not ideal either: Since each run generates a new shared nonce, and we can have infinitely many runs, we have a situation where the two parties sharing a counter have an infinite pool of shared knowledge.

5.1.6 Real-World Considerations Regarding Replay-free Channels

In this section, we suggest an alternative to simply declaring channels between a human and a device replay-free, while still fulfilling the requirement of the human having no initial state.

Ensuring uniqueness is not a matter of channels and their configurations, but rather a matter of state. For a message to be replay free, it needs to be absolutely unique, and there must be mechanisms in place to verify that it is absolutely unique. We achieve this by adding unique values such as nonces to an otherwise arbitrary message. The difficulty in replay-free communication between the the device and the human lies in the latter having no initial state. Apart from simply declaring channels between the device and the human to be replay free, the only way to solve this issue is to have the human choose a unique "verification number" that gets repeated to him along with the message. This is, in essence, nothing more than simply agreeing on a temporary state. This solution however has two issues of its own:

First, it requires a two-way secure channel between the device and the human, which is not always given. Second, it still implicitly assumes that the human has state. This is due to the way Tamarin generates nonces: It is guaranteed that the human will never choose the same verification number twice in different runs (and thereby implied that the human has some internal mechanism of deciding whether or not a value has already been used), which is a very unrealistic assumption.

The goal is for the sender and receiver to establish some common point of reference without having to exchange messages. If we can do this, we can simply use the current counter or nonce models to simulate this in Tamarin. The obvious candidate for such a task would be timestamps. A timestamp is readily available, always increases in value, and does not require the human to keep

state between sessions. Also, it would only require finite state on the server's end, to remember the last used timestamp and ensure that subsequent messages have higher values. Obviously, this solution would only work in cases where the human is in the role of the sender, because the receiver must still maintain state. However, this is the case for many scenarios (e.g. voting, where the human must only submit his own vote, but the server must keep track of all the votes).

In conclusion, we propose the following methods as candidates for establishing a replay-free channel between two parties (assuming there is no two-way secure channel between them):

- If both parties have infinite state, the sender can generate a fresh random value (ensuring it has not been used before by comparing it to a list of previously used values), store it, and send it to the receiver, which does the same thing.
- If either or both parties have finite state, uniqueness is created by a "counter", a value that is always increased with each message. The receiver verifies that the received value is always strictly larger than its currently saved value.
- If the sender has no (initial) state, a common reference point must be created between the receiver and the sender. The most likely candidate for this task are timestamps.

5.2 Characterization with Respect to Channel Use and Ordering

5.2.1 Channel Use

As an additional task, we tried to determine if there is a unique order in which available channels must be used to enable a successful protocol. To do this, we first try to find a minimal protocol for every topology.

A protocol in any of the proposed topologies must fulfill a certain requirement given by the topology, e.g. providing a originating secure channel from H to S. A protocol of minimal length must obviously terminate as soon as this requirement is fulfilled. We can now try to characterize both the length and the form of such a minimal protocol using the following lemmas.

In a first step, we attempt to determine the minimal length of such a protocol in each of the minimal topologies described in [5].

Lemma 1. *Let $\mathcal{S}_x$ be the set of all HISP topologies for which there is a protocol satisfying a property x (e.g. an authentic channel from H to S). For every minimal HISP topology $\tau \in \mathcal{S}_x$, every protocol fulfilling x must use every edge in τ at least once.*

Proof. Let τ be a minimal HISP topology. Assume there is a successful protocol on τ that does not use all edges. This implies that τ is not minimal, which is a contradiction to the minimality of τ. □

Now that we have a lower bound for the minimal length, we can additionally restrict the form of a minimal protocol:

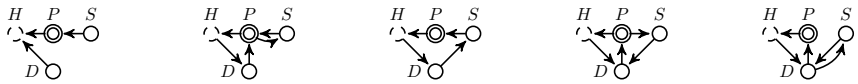
Lemma 2. *Let τ be a HISP topology for which there exist protocols providing a channel from a sender X to a receiver Y. A protocol of minimal length which provides this channel must end at party Y.*

Proof. Assume there is a minimal protocol that does not end at the party Y. Let the set of steps performed in the protocol (channel usages, knowledge exchange, etc.) after Y was last reached be denoted as S. Since Y is never reached within S, Y does not gain any additional knowledge during the execution of the steps in S. Because we assume the protocol fulfills its given requirement, Y must then already have the required knowledge before any of the steps in S are performed. This means that any steps performed in S are redundant and since we require that the protocol be minimal in length, S must therefore be empty. □

Using these two lemmas, we can now describe the form of any minimal protocol more precisely: Any Euler path in a topology ending at the receiver is a candidate for allowing a minimal protocol. Obviously, if such a path does not exist or fulfill the requirements, additional analysis is necessary. In most cases however, the Euler path actually yields a valid protocol.

The following is an analysis of all topologies described in [5], where we determine the minimal protocol for each topology. Many topologies not only have a unique Euler path, but also have trivial minimal protocols that follow the Euler path of the topology and are thus omitted from this section. Please see [5] for the full set of protocols. In the following, only topologies with protocols that do not follow an Euler path are covered. Also note that in the following, the enumeration used for the theorems and lemmas corresponds to the one used in [5].

Theorem 6.
Requirements: Secure channel from S to H.



(a) (d) (e) (f) (g)

a. This topology contains no Euler path. However, it is still possible to create a minimal protocol which traverses every edge only once. We use the protocol described in [5]: S simply sends the code representing the message it wants to communicate via P to H. H receives all predefined message/code pairs from D and can determine the message that S wanted to communicate.

d. This topology consists of an Euler cycle, hence there are many ways to create paths covering each edge exactly once. Only one of these paths ends at H however, which is required by Lemma 2. Therefore, the only way to traverse all edges while still fulfilling the lemmas is using the path $(H, D), (D, P), (P, S), (S, P), (P, H)$. The protocol using this path is described in [5], p.20 (listed as Lemma 8a).

e. As before, this topology contains multiple Euler paths, with only one ending at H, namely $(H, D), (D, S), (S, P), (P, H)$. The minimal protocol thus uses the same approach as before, and is described in [5].

f. This topology contains no Euler path ending at H, which is a requirement for a minimal protocol according to Lemma 2. This means that a minimal protocol for this topology must use one channel at least twice. The minimal protocol described in [5] uses the edge (D, P) twice and does the following: H selects multiple message/code pairs as possible answers of

S and sends them to S via D and P. S chooses which message it wants to relay to H and sends the corresponding code back via D and P.

g. This topology again has many Euler paths, with only one ending at H, namely $(H, D), (D, S), (S, D), (D, P), (P, H)$. The protocol is again described in [5].

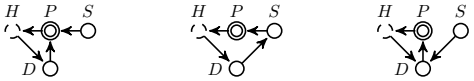
Lemma 5.
Requirements: Secure channel from S to H.

(a) (d)

a. This topology contains no Euler path. The minimal protocol will let S send the code of the message it wants to communicate via P to H. H will then retrieve all message/code pairs from D, and compare the codes to determine the message chosen by S.

d. This topology is not minimal (as stated in [5]). The minimal protocol simply consists of S sending the message securely to H via D.

Lemma 7.
Requirements: Originating authentic channel from S to H.



(a) (b) (c)

None of these topologies have a successful protocol that uses every edge at most once. We will argue this on a case by case basis. All protocols will however have the following in common: Since H is incapable of performing any cryptographic operations, the authentic message from S must first pass through D to verify its authenticity. Additionally, H must be convinced of the authenticity of this message, so there must be knowledge transferred from D to H after the validation has taken place. This requirement will help us define the form of these protocols.

a. The most direct route for a signed message sent by S to get validated is by sending it via P to H, and then to D. From there, the most direct route to send a confirmation to H is via P. Hence, the minimal protocol looks as follows: S sends the signed message to H via P. H then generates a nonce, and sends the signed message and the nonce to D. D validates the authenticity of the message, and if successful, sends the nonce back to H via P.

b. The most direct route for a signed message sent by S to get validated is by sending it via P to H, and then to D. The most direct route to send a message back to H now involves making a loop via S and P. The minimal protocol is therefore similar to the previous one, except that the nonce is sent back to H via S, then P.

c. Here, S can send the message directly to D to be verified. However, this means that H has not yet seen the message that was verified (which was not the case in the previous topologies). The problem here is not for D to verify the authenticity of a message, but for D to verify the authenticity of a message that H has *seen*. As such, D must first send the message to H via P, and then verify the message that H sends back. This prevents P from modifying the message that H sees after it has been verified by D. Finally, D can prove authenticity of the message to H as described above using the nonce method.

5.2.2 Channel Ordering

Now that we have determined minimal protocols for all topologies, we can analyse the order of channel usages.

It is impossible to claim that a certain protocol must start or end with a certain channel, since it is possible to take any successful protocol, and simply extend it at any point with additional channel usage. For example, for a topology τ containing edges $\{(H, D), (D, P), (P, S)\}$, a valid protocol might traverse the edges as so: $(H, D), (D, P), (P, S), (P, S), (P, S)$ (a possible explanation for this could be e.g. that the platform transmits messages in parts). Therefore, we will instead make statements about the minimal chain of channel uses necessary in order to achieve a successful protocol (in the above case, this would obviously be $(H, D), (D, P), (P, S)$).

The analysis of any topology containing a unique Euler path (ending at the recipient designated by the topology requirement) is trivial: Since the Euler path is unique, the minimal order in which the channels must be used corresponds to the order of edges in the path. Therefore, we will only discuss topologies where there is no unique Euler path present. As before, note that the enumeration used for the theorems and lemmas corresponds to the one used in [5].

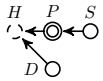
Theorem 6.

(a) (f)

a. This topology requires multiple paths to traverse all edges. This means that there is no unique minimal chain, since these events can be performed at arbitrary times. The minimal chains for this topology are therefore all variations of the chain $(S, P), (P, H)$ combined with the edge (D, H).

f. Since this topology contains no Euler path ending at H, any minimal protocol must use at least one edge twice. Observe that the only incoming node to H comes from P, which means that it is not possible for H to learn a fresh message coming from S (since P learns any message that H learns). The only way to securely send a message in this topology is by having H securely send a codebook to S first. The unique minimal chain for this operation is $(H, D), (D, P), (P, S)$. Once S has this codebook, it can communicate a message to H by sending the corresponding code. The unique minimal chain for this operation is $(S, D), (D, P), (P, H)$. Combining these two chains gives us the minimal chain $(H, D), (D, P), (P, S), (S, D), (D, P), (P, H)$.

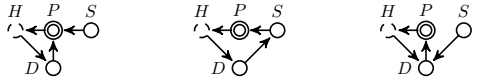
Lemma 5.



(a)

a. This topology requires multiple paths to traverse all edges. The minimal chains are same as in *Theorem 6a*, all variations of the chain $(S, P), (P, H)$ combined with the edge (D, H).

Lemma 7.
As stated previously, we know that since H has limited computing capabilities, any message for H must first be verified as authentic by D. Additionally, this information must then again be passed to H. We can use these facts and Lemma 2 to infer the uniqueness of the following three minimal chains.



(a) (b) (c)

a. The first step in a successful protocol must be to have the message from S validated by D, and then send the validation to H. Based on the minimal protocol described previously, this means that the minimal chain is $(S,P),(P,H),(H,D),(D,P),(P,H)$.

b. Message verification happens by having S to send the message to D via P and H, and then sending that verification back to H via another loop. Hence, the minimal chain is $(S,P),(P,H),(H,D),(D,S),(S,P),(P,H)$.

c. As stated before, H must see the message first and then send whatever it sees to S. Again, based on the minimal protocol suggested previously, the minimal chain is $(S,D),(D,P),(P,H),(H,D),(D,P),(P,H)$.

6 Case Study: Swiss E-Voting Protocol

In this section, we apply all results and gained knowledge from previous sections to a real-world problem, namely a draft version of a candidate protocol for Swiss E-Voting proposed by the task group *Homologation Vote électronique*, which is currently not publicly available.

6.1 The Proposed Protocol

The report published by *Homologation Vote électronique* contains a specification of a potential E-Voting protocol, which we will describe in this section.

6.1.1 Protocol Components

Control Components
These devices contain parts of a secret key and are responsible for distributed creation of code lists, mixing, decrypting and tallying received votes, and generating cryptographic proofs.

Auditors
Auditors are responsible for ensuring the correctness of the voting process. This includes verifying the proofs created by the control components, ensuring the adherence to vote secrecy, and ensuring the correctness of voting results.

Voter Identification Card
This card contains a 16-digit code that is used to identify the voter. It is sent together with the

code list via mail, which we assume to be a secure channel outside the scope of the protocol we will model.

Code List
This list contains a unique ID, one control code per candidate, a confirmation code, and a finalization code. Every list is distributed to a voter in a secure manner.

Printing Press
The press is responsible for taking the data provided by the control components and creating the physical code lists and voter identification cards. This entity is run separately from the voting system.

Voting System
The voting system is the server that the voter communicates with. It handles the authentication of the user and interacts with the control components to verify votes and return the correct candidate and finalization codes.

6.1.2 The Voting Process

First, the control components generate the necessary data (codes, IDs, etc.). This data is then sent to the printing press, which generates the lists, as well as to the voting system. This is done in an anonymized manner, such that the voting system cannot associate codes with voters. This allows the voting system to verify the correctness of the codes without breaking voter confidentiality. The exact process is involved, and will be omitted here.

We will describe in full detail the actual interaction between the voter and the voting system, since that is what our protocol model will be based on. Any other aspects of the voting process are outside the scope of this analysis.

The interaction of voter and voting system is as follows:

- The voter logs in to the voting system by providing the code on his Voting Identification Card and his unique list ID, and sends his vote to the voting system.
- The voting system sends the encrypted vote and the list ID to the control components, which "blind" it and send the blinded vote back to the voting system.
- The system now checks its index to find this vote and returns the corresponding control code (which will be the index of the vote) to the voter.
- The voter compares the code he has received to the control code on his list. If the code corresponding to the candidate he chose is the same as the one he has received from the voting system, he will send the confirmation code on his list. Otherwise, he will abort and vote by mail.
- The voting system forwards the confirmation code to the control components. These will verify the code, and send the finalization code to the voting system.
- The voting system sends the finalization code to the voter, who compares it to the code on his list. If successful, the voting process is complete. Otherwise, the voter must contact a helpdesk.

6.2 The Model

6.2.1 Alice & Bob Specification

Our model focuses only on the voting aspect of the process, we assume that the preparation steps have all been completed successfully. The full protocol model can be found at [6]. Here, we will only present the shorthand notation of our proposed protocol model. Note that this notation introduces the new symbol $\bullet\Rightarrow\bullet$ representing a secure, replay-free channel, which is essential for proving injective agreement:

Protocol Swiss_Evoting

$$
\begin{aligned}
&H : \text{knows}(\langle voteID, candidate, D, S\rangle)\\
&D : \text{knows}(\langle listID, candidate, additionalCode, code, finalCode, confCode, H, S\rangle)\\
&S : \text{knows}(\langle voteID, listID, candidate, additionalCode, code, finalCode, confCode, H, D\rangle)\\
D \bullet\Rightarrow\bullet\ H &: \langle listID, additionalCode, Code, candidate, finalCode, confCode, S\rangle\\
H \circ\!\!\rightarrow\!\!\circ\ P &: \langle S, voteID, listID, additionalCode\rangle\\
P \circ\!\!\rightarrow\!\!\circ\ S &: \langle voteID, listID, additionalCode\rangle\\
S \circ\!\!\rightarrow\!\!\circ\ P &: code\\
P \circ\!\!\rightarrow\!\!\circ\ H &: code\\
H \circ\!\!\rightarrow\!\!\circ\ P &: confCode\\
P \circ\!\!\rightarrow\!\!\circ\ S &: confCode\\
S \circ\!\!\rightarrow\!\!\circ\ P &: finalCode\\
P \circ\!\!\rightarrow\!\!\circ\ H &: finalCode
\end{aligned}
$$

6.2.2 Design decisions

While we try to model our protocol as close as possible to the guidelines proposed in the report, there are still some deviations and simplifications we make to allow for a more concise and easier verifiable protocol. The full protocol model can be found at [6]. Here, we only list the design decisions made for our model.

- We allow H to have some initial knowledge, namely his voteID (learned beforehand from the identification card), his chosen candidate, and the identities of D and S (since the server identity should be publicly known, such as a government-operated voting site).
- We add S to the initial knowledge of D to prevent a mixup in the first protocol rule, where D needs to send data to H. This modification is the equivalent of printing e.g. the URL of the voting website on the code list.
- We decide to maintain voting secrecy by adding one additional code per candidate to the code list, which is sent to the server instead of the unencrypted candidate name suggested in the paper. This code is a hash of the candidate name and a fresh value, and is generated per-user in the setup rule. While this means our model deviates from the original specification by providing stronger guarantees, we believe that voting secrecy is equally important to other properties such as correctness and authenticity.

- The candidate name is sent along in the first message transmitted by D to ensure that H gets the code for the correct candidate, i.e. the one chosen in the setup rule.
- We define the channel between D and H as secure and replay-free. We justify this by the fact that the code list is delivered securely to the voter, and because we assume the voter uses it in an environment where the adversary cannot learn or modify the contents of the list (e.g. inside the voter's home).
- The implementation in the report describes a login process where the voter must first submit his voter ID as well as other information before he is authenticated and allowed to cast his vote. We have simplified this process in our model by omitting the login acknowledgement sent by the server; the voter now sends his voteID and the code corresponding to his chosen candidate in one message.

6.2.3 Candidate Creation

A simple way to create candidates would be to define them as nonces for each individual run. However, we wanted a more realistic model of candidates, adhering to the following premises:

- Candidates are determined prior to any voting. Once voting starts, the number and identity of candidates must not change.
- There is no restriction on how many candidates can be added to the voting list.
- All candidate names must be public knowledge.

Based on these requirements, we created two possible methods for candidate creation, detailed below:

Candidate Facts
This method creates candidates by storing them in persistent facts, allowing reuse between different runs. We can also choose between the ability to generate an unlimited amount of candidates, or set a strict, hardcoded limit. Rule 21 shows the creation of candidates using the unlimited method.

$$[\mathsf{Fr}(\sim candidate)] -[\ \mathsf{CreateCandidate}(\sim candidate)\ ]\rightarrow [!\mathsf{Candidate}(\sim candidate), \mathsf{Out}(\sim candidate)] \tag{21}$$

Since a fresh nonce is created every time this rule is executed, we can have an unlimited number of unique candidates. The Out fact ensures that the adversary learns the names of the created candidates (since we assume that these are publicly known values).

To restrict the total number of candidates, we use predefined public values in the rule as opposed to generating a fresh nonce every time. This ensures that even if the rule is executed multiple times, there will only be a fixed number of candidates. This rule is quite simple, and only generates candidates in the conclusion. For example, Rule 22 shows the generation of two candidates:

$$[\,] -[\ ]\rightarrow [!\mathsf{Candidate}('Candidate1'), !\mathsf{Candidate}('Candidate2')] \tag{22}$$

Note that we do not need to include any Out facts in this rule since the candidates are modeled with public values, and are hence already known to the adversary.

Since both methods produce the same persistent fact Candidate, we can assign the candidate that H will use in the protocol by using the fact Candidate($candidate$) as a premise in the setup rule, regardless of what method was used.

Candidate Lists

An alternative to having multiple facts containing candidate names is to create a multiset that holds all the created values. Again, we have the option of generating an unlimited vs. a limited number of facts. Since we use the multiset rules built into Tamarin, we must preface all protocols using these methods with the code `builtins: multiset`.

The unlimited variant will again create nonces as candidates, and requires three separate rules. The first rule initializes the list with a public value:

$$[\,]\,\text{–}[\ \mathsf{InitList}()\ ]\!\rightarrow [\mathsf{List}('buffer')]$$

We ensure that this rule is only called once (so that we don't have different concurrent lists) with an axiom:

```
axiom one_list:
"All #i #j.  InitList() @i & InitList() @j ==> #i=#j"
```

Next, we can populate the list by calling the following creation function:

$$[\mathsf{Fr}(\sim c), \mathsf{List}(x)]\,\text{–}[\ \mathsf{CreateCandidate}(\ c)\ ]\!\rightarrow [\mathsf{List}(x + \sim c), \mathsf{Out}(\sim c)]$$

This rule adds a new value to the multiset by using Tamarin's builtin multiset operator '+'. Note that the List fact is linear, which ensures that there is only one multiset/list at all times.

As a final step, we fix our list so that no further candidates can be added. This is done with the following rule:

$$[\mathsf{List}(x)]\,\text{–}[\ ]\!\rightarrow [!\mathsf{Candidates}(x)]$$

We now want to allow Tamarin to arbitrarily choose a value from the multiset contained in the Candidates fact as the candidate value for H. This is done by modifying the setup rule as follows:

$$[!\mathsf{Candidates}(x + \sim candidate), \ldots]\,\text{–}[\ \mathsf{Setup}()\ ]\!\rightarrow [\mathsf{AgentState}(\$H,' H\_0', \langle \sim candidate, \ldots \rangle, \sim tid\_H)]$$

This rule allows Tamarin to arbitarily split the multiset into one fresh value ($\sim candidate$) which will be used by H, and some residue (x).

For the limited variant, we reverse the previous approach by swapping the places of the public and fresh values. We only require the Initialize rule, which will look as follows:

$$[\mathsf{Fr}(\sim buffer)]\,\text{–}[\ ]\!\rightarrow [!\mathsf{Candidate}(\sim buffer +' candidate1' +' candidate2')]$$

Note that we can ommit the Out facts since public values are always known to the adversary.

The setup rule must be adjusted in a similar manner:

$$[!\mathsf{Candidates}(x + \$candidate), \ldots]\,\text{–}[\ \mathsf{Setup}()\ ]\!\rightarrow [\mathsf{AgentState}(\$H,' H\_0', \langle \$candidate, \ldots \rangle, \sim tid\_H)]$$

Ensuring Invariance
We want to guarantee that a pool of available voters does not change while the vote itself is in progress. To do this, we need to restrict the time during which candidates can be created. The list method already covers this requirement by having a rule to fix the candidate list and an axiom for ensuring that there is only one list in use. Additionally, the limited variant of the fact method also does not have this problem, since calling the candidate creation rules will not change the available candidate pool. For the unlimited fact variant however, we can make use of the CreateCandidate action with the following axiom:

```
axiom restrict_list_creation:
"All c #i #j.  CreateCandidate(c) @i & Setup() @j ==> #i<#j"
```

With this axiom, we can restrict the execution of all candidate creation rules to before the first execution of the setup rule, thus ensuring that candidates can only be added before the voting protocol starts.

6.3 Analysis

6.3.1 Server Commit

In this step, our goal is to verify that it is not possible for a malicious voter to have his vote counted multiple times by the voting system. This would obviously be a replay attack, and hence we choose injective agreement as the property to verify. Since the server needs to be secured against replay attacks on data sent by voters, we choose the server to be the committing party and add the Running action to H and the Commit action to S. Note that we omit the protocol models used in this analysis. The full protocol models (including proofs for the various properties) can be found at [6].

We analyse protocol variants using all four proposed methods of candidate creation (limited/unlimited, facts/list). Table 3 shows an overview of the results:

Protocol Variant	**Fulfills NIA**	**Fulfills IA**
Limited List	Yes	Yes, if channel (D,H) declared replay free
Limited Facts	Yes	Yes, if channel (D,H) declared replay free
Unlimited List	Unknown	Unknown
Unlimited Facts	Yes	Yes, if channel (D,H) declared replay free

Table 3: Results for server-commit analysis of the proposed Swiss E-Voting protocol.

Note: Proofs for the unlimited list variants fail because Tamarin does indeed try to create an unlimited list (and thus gets stuck in an endless loop). For some reason, the same phenomenon does not occur for the unlimited fact variant, even though the same possibilities are given.

6.3.2 Human Commit

In this section, we want to reverse the previous analysis by checking if it is possible for the adversary to convince the voter that his vote has been counted, when in fact it had been discarded or altered. We do this by adding the Running action to S and the Commit action to H, where S performs the

Running action directly after it learns which candidate H has supposedly voted for and H commits after receiving the finalization code.

Note that the property we are verifying here is not the absence of replay attacks, but rather the agreement of S and H on the candidate that H has voted for. Hence, we choose non-injective agreement as the property to verify, with the idea that we only need to check if H has committed to a value that S did not send (and therefore has no corresponding Running action).

We found that all variants of the protocol are vulnerable to an attack on non-injective agreement, although this attack is not very practical.

Illustration of the Attack on NIA

Consider the following scenario: A household has 5 people which are eligible to vote. Since politics are often discussed, it is known to everyone in that household that 4 people will vote for party A, while one person (the adversary) will vote for party B.

After receiving the required voting documents, the adversary simply photocopies his own voting list 4 times and exchanges everyone else's legitimate voting list with his own (note that this is not an illegal action according to our model, since the lists are still honest trusted devices. The adversary gains no knowledge, nor does he replay any information. He simply exchanges the lists without looking at them).

Before anyone else votes, the adversary now votes for party A, and plants a trojan on every PC in the household, which e.g. exchanges any login attempt to the voting site with his own credentials (assuming one can change a vote arbitrarily often before voting is closed).

As a result, all members in the household will vote in the adversary's name, not realizing that their own vote is not counted. Therefore, the final votes will be A=1,B=0 or even A=0,B=1 if the adversary votes again after everyone is done. The correct votes should however be A=4,B=1, so party A has lost at least 3 votes through this attack.

Because this attack requires physical access to the lists of all the voters, it might be regarded as impractical, especially since an adversary could do a lot more damage than just exchanging lists (i.e. simply vote for another candidate under the voter's name later).

Cause of the NIA Attack & Solution

The cause of the attack is simple: The list containing all candidates and control codes does not have the voting ID of the intended recipient printed on it. As such, it is easy for anyone with physical access to these lists to exchange them, causing all affected voters to vote as the adversary (without them noticing), thus causing a skew in voting results.

The solution is also simple: If the voter ID is added to the list, this attack is no longer possible (if we assume that the voter cross-checks his voter ID with the ID printed on the list). Implementing this in Tamarin (by adding the voter ID to the initial knowledge of D and sending it along in the message to H), allowed Tamarin to verify both NIA and IA properties for almost all variants of the protocol (again with the exception of the unlimited list variant, for which Tamarin does not terminate). To reiterate, Table 4 shows all results for the human-commit analysis:

Protocol Variant	**Fulfills** NIA	**Fulfills IA**
Limited List (unmodified)	No	No
Limited Facts (unmodified)	No	No
Unlimited List (unmodified)	Unknown	Unknown
Unlimited Facts (unmodified)	No	No
Limited List (added voter ID)	Yes	Yes
Limited Facts (added voter ID)	Yes	Yes
Unlimited List (added voter ID)	Unknown	Unknown
Unlimited Facts (added voter ID)	Yes	Yes

Table 4: Results for server-commit analysis of the proposed Swiss E-Voting protocol.

7 Conclusions

The goal of this thesis was to extend the HISP model by providing additional channel and security properties to more precisely model real-world protocols concerning the communication between a human and a remote server. In particular, we wanted the ability to formulate strong authenticity properties for the existing models. We have achieved this goal by formalizing availability, non-replayability and instantaneity as channel assumptions, as well as formalizing the security properties of aliveness, recentness, (non-)injective agreement and (non-)injective synchronization.

We applied these new properties to the topologies described in [5]. We showed that non-injective agreement holds for all protocol models without having to modify the underlying topologies, and injective agreement holds for almost all protocol models with only minimal changes to the topologies. Furthermore, we have shown the relevance of these model extensions by modeling and analyzing an electronic voting protocol proposed by a task group of the Swiss government. The flaw found in this protocol model demonstrates both the necessity of having ways to express these security properties, as well as the necessity of proof checking protocols in general, since even a small oversight like not including an ID number on a list can lead to attacks and insecure protocols.

During the course of this thesis, additional open questions have appeared that might be the focus of future research. One of these concerns characterization of topologies with respect to weaker authentication properties. Is it possible to create minimal topologies that are smaller than the current ones, but only fulfill, e.g., aliveness? The other question concerns the expansion of the HISP model to contain additional nodes, such as multiple devices, servers or platforms. In the context of the E-Voting protocol model, we could e.g introduce an additional device node to model the voter identification card, or model the voting system and control components separately as two servers. Another option could be to model the election authority in more detail, e.g. explicitly modeling the bulletin board or the auditing process.

8 References

[1] C. Cremers. *The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols*, 2008.

[2] ProVerif Homepage, September 2014.
`http://prosecco.gforge.inria.fr/personal/bblanche/proverif/`

[3] B. Schmidt, S. Meier, C. Cremers and D. Basin. *Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties*, June 2012.

[4] G. Lowe. *A Hierarchy of Authentication Specifications*. In *CSFW '97 Proceedings of the 10th IEEE Workshop on Computer Security Foundations*, page 31.

[5] D. Basin, M. Schläpfer, and S. Radomirović. *A Complete Characterization of Secure Human-Server Communication*, 2014.

[6] Repository of HISP models for the thesis *E-voting and Secure Human-Server Communication*, September 2014.
`http://www.infsec.ethz.ch/hisp`

[7] Repository of the Tamarin prover, September 2014.
`https://github.com/tamarin-prover/tamarin-prover`

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):

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

Name(s):	**First name(s):**

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)**

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