Formal Verification of Attack Simplification Lemmas

Author:
Cornelissen, Eric
eric.cornelissen@ut.ee

Supervisor:
Laur, Sven
sven.laur@ut.ee

December 20, 2019

ABSTRACT Cryptographic analysis is a tedious process and formal verification tools were introduced to alleviate some of the difficulty. Unfortunately, formal methods cannot be applied directly to existing cryptographic proofs as mismatches in the translations may result in errors. Work by Pfitzmann and Waidner on a model for asynchronous reactive systems aimed to provide a bridge between the two approaches. However, this model is subject to irrelevant timing attacks. New work by Laur et al. aims to show that, under natural assumptions, many time-sensitive details can be abstracted away. This paper describes an attempt to formally verify part of the work by Laur et al. using the Coq Proof Assistant.

1 Introduction

Cryptographic analysis by hand is difficult, tedious, and error-prone. Time and time again it has been shown that seemingly secure protocols have major flaws. It is not uncommon that such flaws are discovered years later only after the protocol has been implemented and is used in practice. Moreover, the recent focus on more exotic secure protocols - or more generally secure reactive systems - such as secure multiparty computation are simply too difficult to prove by hand.

A reactive system *must* have proper security proof before being used in practice, and today it is an accepted fact within the security community that the rigorous handwritten approach does not guarantee security. Instead, formal verification systems are used more and more to support proofs. Unfortunately, there is no clear translation between handwritten proofs and proofs from formal methods.

The seminal work by Pfitzmann and Waidner [3] aims to provide a bridging model between the handwritten approach and formal verification systems. Their model consists of two layers: the first layer which retains a view of the underlying cryptographic components of a reactive system, and a second layer

which provides abstract services that hide the cryptographic details. The former can be used for rigorous handwritten proofs, while the latter is perfect to implement in formal verification systems.

However, Pfitzmann and Waidner's model is subject to "irrelevant" timing attacks and researchers using the model must go to great lengths to show that their proof also holds for these attacks. This problem affects the analysis of Multi-Party Computation Protocols in particular. Some attacks can be considered "irrelevant" because they model behaviour that does not help the adversary gain an advantage in the protocol. New work by Laur et al. (still in development) aims to show that, under certain natural assumptions about the adversary, a proof in Pfitzmann and Waidner's model implies security regardless of such timing attacks.

In this work one of the lemmas from the work by Laur et al. is proved using the Coq Proof Assistant. Coq is a proof management system that provides a formal language to write mathematical definitions as well as theorems. Theorems can be proven with Coq through a semi-interactive development environment to obtain machine-checked proofs [2].

This paper is organized as follows. Section 2 provides preliminary details (notation and definitions) and Section 3 gives background information on Pfitzmann and Waidner model, Laur's work, and the Coq Proof Assistant. Then, Section 4 describes how the two models have been translated into Coq and how the lemmas are proven. Finally, Section 5 and 6 provide a discussion on the approach taken in this paper and conclusion respectively.

2 Preliminaries

Notation Π is used to denote a protocol. For Pfitzmann and Waidner's model we denote by $\mathcal{A}$ the adversary, $\mathcal{P}_i$ a party in a protocol Π, $\mathcal{I}_i$ the component of $\mathcal{P}_i$ that executes a protocol Π faithfully, and $\mathcal{Z}_i$ the component of $\mathcal{P}_i$ that $\mathcal{A}$ uses to corrupt $\mathcal{P}_i$. $\mathcal{F}_j$ represent ideal functionality for the protocol Π. A buffer is denoted as b^+ or b^- depending on its direction (outgoing and incoming respectively).

A subscript is used for indexing, so $\mathcal{P}_i$, $\mathcal{I}_i$, $\mathcal{Z}_i$, and $\mathcal{F}_j$ (where $\mathcal{P}_i = (\mathcal{I}_i, \mathcal{Z}_i)$) are specifically the ith/jth instance of each of the respective components. Similarly, b_k^+ and b_k^- are the kth buffer. Morever, b_k^+ and b_k^- denote a connected pair of buffers, whereas b_k^+ and b_l^- denote two unrelated buffers.

A `monospace` font is used to denote Coq code.

Definitions The following contains informal definitions of terms used throughout the paper. A *Protocol* or *Asynchronous Reactive System* is a system that works asynchronously, i.e. parties do not communicate in real-time and is reactive, i.e. parties (can) provide many inputs into the system. In the context of the use of cryptographic protocols in the real world the model captures that message exchange is not instant (asynchronous) and that protocol consists of multiple messages or even multiple subprotocols (reactive).

An *Adversary* is an entity that is malicious and tries to either distort the workings of a protocol or subvert the security of a protocol. A *(Honest) Party* is an entity that participates in a protocol according to the protocol specifications, whereas a *Corrupt(ed) Party* is an entity that participates in a protocol as the adversary wishes. *Ideal Functionalities* make up the public functionality of a protocol that always behaves as it should (i.e. it cannot be controlled by the adversary entirely unless $\mathcal{A}$ controls all inputs into the ideal functionality).

3 Background

This section introduces Pfitzmann and Waidner model for Asynchronous Reactive Systems [3], explains the parts of Laur's work that are relevant to the present work, and finally, it introduces the Coq Proof Assistant.

3.1 A Model for Asynchronous Reactive Systems

Pfitzmann and Waidner seminal work [3] describes a novel modelling technique for Asynchronous Reactive Systems. Their model has cryptographically sound semantics and supports the composition of systems. As a result, it can be used as a bridge between rigorous hand-written proofs and tool-supported formal verification techniques. The model follows the common approach in cryptography to use simulations, and should, therefore, be familiar to cryptographers.

What follows is a condensed description of the model, for a full treatment of the model we refer to the original paper by Pfitzmann and Waidner [3]. For this paper, the model consists of four principal components and connections (in the form of *buffers*) connecting them. This setup is depicted in Figure 1 for a 5-party protocol with one party displayed in full.

The components are: 1) the *Interpreter* $\mathcal{I}_i$ which executes the protocol faithfully; 2) the *Corruptor* (or *Corruption Module*) $\mathcal{Z}_i$ which, when activated, gives the adversary full control over the message of the respective $\mathcal{I}_i$; 3) the *Adversary* $\mathcal{A}$ which, as usual, tries to manipulate the protocol to obtain some sort of advantage; and 4) the ideal functionalities $\mathcal{F}$ which compute functions upon receiving inputs. The combination of the $\mathcal{I}_i$ and $\mathcal{Z}_i$ components give rise to a party $\mathcal{P}_i$. The adversary $\mathcal{A}$ may or may not control any party $\mathcal{P}_i$ through $\mathcal{Z}_i$.

The different components can communicate through means of *buffers*. A buffer is a unidirectional communication channel which must be *clocked* to get the message from the sender to the receiver. A message can be any valid message within the protocol being executed or a *query* by the adversary. A query is a message directed at a corrupted parties interpreter $\mathcal{I}_i$ and can be used by $\mathcal{A}$ to obtain information about the internal state of the party.

3.2 Laur's work

The (unpublished) work of Laur addresses the limitation of Pfitzmann and Waidner's model that it is subject to "irrelevant" timing attacks. This manifests

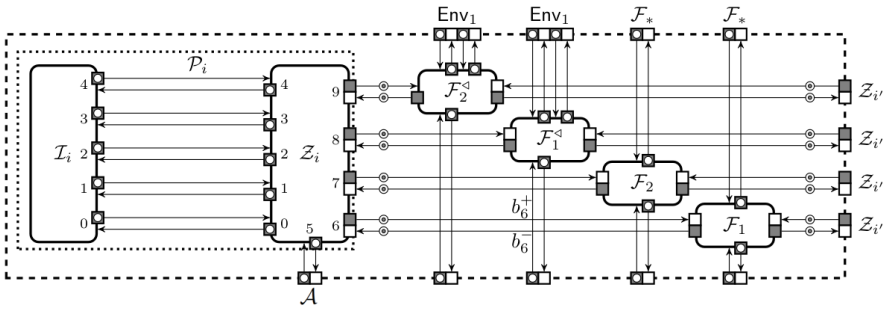

Figure 1: An overview of the components in Pfitzmann and Waidner's model. Note that only a single party $\mathcal{P}_i$ is shown, the other parties live at the far right, where the other $\mathcal{Z}_i$s are depicted. (*source: Laur et al.*)

itself through the fact that $\mathcal{A}$ can advance the protocol throughout the model at any time. As a result, researchers using the model have to prove that their findings also hold when accounting for these edge cases. Laur et al. put certain natural restrictions on the adversary in the original model to obtain an alternative model where proofs entail security even when accounting for "irrelevant" timing attacks (without having to prove that separately).

For the present work one needs to understand Pfitzmann and Waidner's model, as explained previously, and the following restrictions and modifications on the model. How these properties give the desired result is out of the scope of this paper and will not be discussed.

$\mathcal{I}_i$ must be a *Well-Formed Program*. This entails, among others, that it does not alter its memory before being engaged in a protocol Π, nor does it try to end a protocol more than once. When combining the previous with observations that are omitted here for brevity, any adversary in the model can be reduced to an adversary that adheres to a *Tight Message Schedule*. This means that the adversary will never send two consecutive messages from a party to a $\mathcal{F}_j$ or vice versa. $\mathcal{F}_j$ follows strict behaviour rules. This entails, among others, that it only works with inputs it expects, computes values immediately upon having received all inputs, and tags its messages correctly.

Next is the definition of a *Semi-Simplistic Adversary*, which, together with the previous restrictions and modifications as axioms, form the basis for the lemmas that will be proven with Coq. The structure of a Semi-Simplistic Adversary $\mathcal{A}'$ in relation to the other components is also shown in Figure 2.

1. The adversary can clock outgoing buffers b^+ of parties only when all incoming buffers b^- to all corrupted parties are empty.

2. The adversary can clock any incoming buffers b^- to an honest party only when all incoming buffers b^- to all corrupted parties are empty.

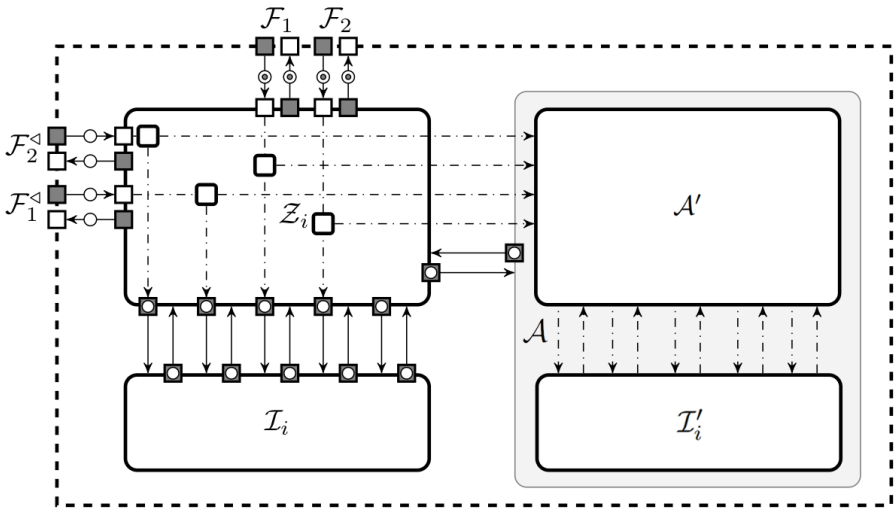

Figure 2: An overview of the components in the model after the modifications by Laur et al. Note that the adversary, in contrast to Figure 1 is now a composition of an $\mathcal{A}$ and a copy of $\mathcal{I}_i$. (*source: Laur et al.*)

3. Upon receiving a message from a $\mathcal{F}_j$, the adversary immediately forwards it to the respective $\mathcal{I}_i$.
4. The adversary can send any message to a $\mathcal{F}_j$.
5. The adversary can always send corruption instruction to $\mathcal{I}$ that do not alter the state of $\mathcal{I}$.
6. The adversary performs no other corrupt actions.

3.3 Coq Proof Assistant

The Coq Proof Assistant[1] is, "a formal proof management system [that] provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs." [1]

This means it can be used to construct machine-checked proofs through a formal language. This language can also be used to write mathematical definitions and executable algorithms, i.e. it is a (Turing incomplete) programming language. One can think of Coq as a system that can be used to specify a formally defined system through mathematical expressions - typically programming languages [1], but in this case a cryptographic model - and then prove lemmas and theorems about the system.

[1]See https://coq.inria.fr/.

Coq comes with "an environment for semi-interactive [...] machine-checked proofs". *Machine-checked* means the proofs for the lemmas and theorems written in Coq are automatically checked for correctness. *Semi-interactive* means that it is possible to step forwards and backwards through a Coq program, which is particularly useful in proofs where each proof-step can be evaluated separately (in contrast to say a function definition which Coq evaluates entirely at once).

To learn more about Coq we recommend the reference manual [2] and the Software Foundations books [5]. Next, some relevant terminology and syntax for this paper are introduced. Coq is a strictly typed functional programming language.

3.3.1 Programming in Coq

The primary aspect of the programming language used in the present work are the `Record`, the `Inductive` definition, and lists.[2] Practically, it is important to note that expressions are ended with a period rather than a semi-colon, and that type annotations in Coq go behind the variable name as in `(b : bool)` or `(n : nat)`. A `Record` is similar to *Struct*s in the C family or *Object*s in object-oriented programming languages but, like a *Struct*, it cannot have "methods".

Lists in Coq use a notation common to functional programming languages. Putting expressions inside angular brackets, `[` and `]`, constructs a list out of the values. The notation `head :: tail` constructs[3] a list where `head` is the first element of the list and `tail` is a list containing the remaining elements of the list. The notation `neck ++ tail` constructs a new list as the concatenation of the two lists `neck` and `tail`.

Records in Coq are collections of values. A new `Record` can be defined using the `Record` keyword followed by the name of the `Record` type, as illustrated in Figure 3. In Figure 3, `mkFoo` can be considered as the *constructor* of the `Record`, which can be used as `(mkFoo true 1)` to create an instance of `Foo`. `n` and `b` are the fields of the `Record` with types `nat` and `bool` respectively. To access the field `b` of an instance of `Foo` one writes `instance.(b)`.

```
Record Foo : Set := mkFoo { b : bool; n : nat }.
```

Figure 3: An example `Record` in Coq.

3.3.2 Proving in Coq

As a proof engine, Coq uses a special syntax through so-called *tactics*. But first, a prover needs to define a theorem or lemma (or corollary). This can be achieved using the keywords `Theorem` or `Lemma` (or `Corollary`) and giving it a

[2] Observant readers may note the syntax `Set` and `Prop` in later sections. It is not important to understand them but the basic intuition is that `Set` is the type of normal types used in programming and `Prop` is the type of logical propositions

[3] Technically it can also deconstruct the list, but this feature is not used in the present work.

name. This is followed by quantification over the variables of interest using the keywords `forall` or `exists`. Finally, the actual theorem can be specified, as illustrated in Figure 4. Once a theorem or lemma is proven the `Qed` keyword must be used to indicate that the proof finished.

```
Theorem my_theorem : forall (l : list nat) (n : nat),
    (len (n :: l)) = (len (l ++ [n]))
Proof. ... Qed.
```

Figure 4: A simple example of a theorem stating that adding a value to the beginning or end of the list results in a list of the same length in Coq. The proof of the theorem is omitted for brevity.

During the proof process in Coq one works on a *goal* within a *context*. The goal is the statement that is actively being proven. After the `Proof` keyword, there is only a single goal. If, for example, the goal goes over all natural numbers than, using induction over the natural numbers, one obtains two goals: one to proof the base case and one to proof the step case. The context is the knowledge the prover can use to proof the goal. After the `Proof` keyword, the context is empty and it is only by using tactics on the goal that a prover can obtain knowledge.

The following Coq tactics are necessary to understand the present work:

`intros` This tactic can be used to introduce universally qualified variables as well as hypothesis into the context. For example, in the proof in Figure 4 one would start by using the tactic `intros l n.` to introduce the list `l` and the number `n` into the context.

`reflexivity` This tactic is used when the goal is an equality or inequality that is satisfied (e.g. `n = n` or `3 > 2` respectively) to tell Coq the goal was proven.

`split` This tactic can be used to split a conjunction into two separate proof goals. For example, if the current goal is `P` $\wedge$ `Q` then `split.` will result in two new goals: one to proof `P` and one to proof `Q`.

`rewrite` This tactic can be used to rewrite something in the goal by the equality from a hypothesis in the context. For example, if the goal is to prove `n = x` and there is a hypothesis `H` in the context that `x = y`, then `rewrite H.` will change the goal to `n = y`.

`(e)apply` This tactic can be used to apply a hypothesis to the goal. For example, if the goal is to prove `n = m` and there is a hypothesis `H` in the context that `n = m`, then `apply H.` will prove the goal. One will need to use `eapply` if there is an existential quantifier in the hypothesis that one wants to apply.

`exists` This tactic can be used to concretize an existentially qualified variable. For example, if the goal is to prove that $\exists$ `n : n > 2` then `exists 3` will change the goal to `3 > 2` (which can then be proven using the `reflexivity` tactic).

4 Proving Equality through Bisimulation

The lemma that we aim to prove is that a Semi-Simplistic Adversary as defined in Section 3.2 can, if desired, have equivalent capabilities to a behaviourally unrestricted adversary in the Pfitzmann and Waidner's model. The exact formulation of this lemma is given in Section 4.4. Proving this lemma shows, in effect, that putting these limitations on $\mathcal{A}$ does not make it less powerful and so using this new model does not result in a lower level of security.

To prove the desired lemma in the Coq Proof Assistant, Pfitzmann and Waidner's model needs to be modelled in Coq. The components of the model (i.e. $\mathcal{I}$, $\mathcal{Z}$, $\mathcal{P}$, $\mathcal{A}$, and $\mathcal{F}$) will be modelled as a `Record`. The clocking of buffers by the adversary will be modelled as a step-relation between two consecutive states of the protocol. I.e. any action the adversary can take will be represented as a relation between the entire state of the protocol before the buffer was clocked and after the buffer was clocked. Also, all possible ways in which the adversary can progress the state of the protocol without clocking a buffer is also modelled as a step-relation.

If the clocking of a buffer results in some computation by a component of the model, the result of the computation is outputted as part of the step. For example, if an ideal functionality $\mathcal{F}_j$ that expects n inputs has previously received $n-1$ inputs, the step that inputs the nth value will automatically trigger $\mathcal{F}_j$ to compute the result and output it to all of its outgoing buffers $b_1^+, \ldots, b_k^+$.

Mathematically, a step-relation can be described as follows, where $\mathbb{A}_{\mathcal{A}}$ is the set of all adversary states for a particular $\mathcal{A}$, $\mathbb{P}_\Pi$ the set of all possible states of all parties in Π, and $\mathbb{F}_\Pi$ the set of all possibe states of all functions in Π. The set of possible states Σ of a specific $\mathcal{A}$ and Π is given by $\Sigma_{\mathcal{A},\Pi} \subset \mathbb{A}_{\mathcal{A}} \times \mathbb{P}_\Pi \times \mathbb{F}_\Pi$. Then, a step-relation is a function over states $\sigma_t, \sigma_{t+1} \in \Sigma_{\mathcal{A},\Pi}$ defined as $s : \sigma_t \mapsto \sigma_{t+1}$.

First, this section provides an overview of the generic design choices independent of the original model and Laur's version. Then, the design of the original model is given. This includes a detailed explanation of one of the step-relations rules and an overview of all step-rules that have been defined. Following that the design of Laur's modified model is given. This consists of an overview of new and modified step-relation rules. Finally, the approach taken in proving the desired lemmas is provided.

4.1 Overall design

This section will briefly highlight some of the design decisions made in modelling the two systems. In doing that it provides background details for the following

two sections, but this section does not contain anything insightful in and of itself.

4.1.1 Utilities

A *Message*, as shown in Figure 5, is a `Record` that carries information over buffers between components. There are essentially two types of messages: *queries* and *non-queries*. In Pfitzmann and Waidner model this distinguishes actual values passed around and queries performed by the adversary on corrupt parties (for brevity the response to a query is also flagged as being a query). This separation is required to model the difference between state-changing and non-state-changing messages explicitly.

A message can also be *corrupt* or *not corrupt*. Messages generated by $\mathcal{A}$ are flagged as being corrupt and all other message are not corrupt. Note that all queries are also corrupt which is guaranteed through an Axiom in Coq.[4] The reason for this separation has to do with the definition of a semi-simplistic adversary and will become apparent later.

Without loss of generality, all messages (value or query) are represented as numbers. One can easily imagine that each protocol message or adversarial query is represented as a number which is passed around by the components in the model and interpreted using a translation table by the components.

```
Record Message : Set := mkMessage
                  { value : nat
                  ; query : bool
                  ; corrupt : bool }.
```

Figure 5: The `Record` representing messages passed around in the model.

A *Buffer*, as shown in Figure 6, is the communication medium in the model as explained in Section 3.1. The list of `buffered` messages represents messages send by the sending component and the list of `released` messages represents the messages received by the receiving component.[5]

[4] In Coq an `Axiom` is essentially a `Theorem` or `Lemma` without a proof.

[5] In the actual Coq model a `Buffer` also has an `is_leaky` flag and `leaked` list of messages, for this paper they are not relevant and therefor omitted.

```
Record Buffer : Set := mkBuffer
          { buffered : list Message
          ; released : list Message }.
```

Figure 6: The `Record` representing buffers (b^+, b^-) in the model. Whether they reside in a list of outgoing `Buffer`s or a list of incoming `Buffer`s determines the direction of the `Buffer`.

State, as shown in Figure 7, is used to model the state of certain components (namely $\mathcal{I}_i$, $\mathcal{F}_j$, and $\mathcal{A}$). As the details of the state are not relevant for the lemmas to be proven in the present work they simply capture the received and send values (but not queries).

```
Record State : Set := mkState
          { received : list nat
          ; send : list nat }.
```

Figure 7: The `Record` representing the state of components in the model. The components that have a state are $\mathcal{A}$, $\mathcal{I}_i$, and $\mathcal{F}_j$.

4.1.2 Model Components

As per Figure 8, an interpreter $\mathcal{I}_i$ is modelled as a `Record` with a `State` and two lists, one for incoming and one for outgoing, of `Buffer`s.

```
Record Interpreter : Set := mkInterpreter
          { i_state : State
          ; i_incoming : list Buffer
          ; i_outgoing : list Buffer }.
```

Figure 8: The `Record` representing an interpreter $\mathcal{I}_i$ in the model.

Similarly, as per Figure 9, the adversary $\mathcal{A}$ is modelled as a `Record` with a `State` and two lists, one for incoming and one for outgoing, of `Buffer`s.[6]

[6]Note that the identifiers have different names (e.g. `i_state` for the adversary and `a_state` for interpreters). This is required in `Coq` to be able to address the fields.

```
Record Adversary : Set := mkAdversary
            { a_state : State
            ; a_incoming : list Buffer
            ; a_outgoing : list Buffer }.
```

Figure 9: The `Record` representing the adversary $\mathcal{A}$ in the model.

Then, for Laur's work as per Section 3.2, a slightly different `Record` is needed where the adversary contains within it a copy of $\mathcal{I}_i$. Shown in Figure 10, such an adversary can be modelled as a new `Record` containing within it an $\mathcal{A}$ `Record` and an $\mathcal{I}_i$ `Record`.

```
Record CompositeAdversary : Set := mkCompositeAdversary
                    { A' : Adversary
                    ; I' : Interpreter }.
```

Figure 10: The `Record` representing the semi-simplistic adversary $\mathcal{A}$ in the model.

A corruptor component $\mathcal{Z}_i$, whose record is shown in Figure 11, is modelled as a boolean value indicating whether the party `is_corrupted` (put differently: if $\mathcal{Z}_i$ is enabled) and two lists, one for incoming and one for outgoing, of `Buffer`s. Note that the details of where the `Buffer`s are going is not modelled explicitly. This is captured in the step-relation by having the details of the `Buffer` be shared uniquely between $\mathcal{Z}$ and the respective component.

```
Record Corruptor : Set := mkCorruptor
            { is_corrupted : bool
            ; z_incoming : list Buffer
            ; z_outgoing : list Buffer }.
```

Figure 11: The `Record` representing a corruptor $\mathcal{Z}_i$ in the model.

Then, from a $\mathcal{I}_i$ and $\mathcal{Z}_i$ a party $\mathcal{P}_i$ is modelled. The $\mathcal{P}_i$ record simply contains a $\mathcal{I}_i$ record and a $\mathcal{Z}_i$ record. This is shown in Figure 12.

```
Record Party : Set := mkParty
          { I : Interpreter
          ; Z : Corruptor }.
```

Figure 12: The `Record` representing a party $\mathcal{P}_i$ in the model.

Similar to the adversary and interpreter `Record`, as shown in Figure 13, an ideal functionality $\mathcal{F}_j$ is modelled as a `Record` with a `State` and two lists, one for incoming and one for outgoing, of `Buffer`s.

```
Record FunctionBox : Set := mkFunctionBox
          { f_state : State
          ; f_incoming : list Buffer
          ; f_outgoing : list Buffer }.
```

Figure 13: The `Record` representing an ideal functionality $\mathcal{F}_j$ in the model.

4.2 Modelling the Original Model

The original model by Pfitzmann and Waidner is represented by a `Record` containing an adversary $\mathcal{A}$, a list of parties $\mathcal{P}$, and a list of ideal functionalities $\mathcal{F}$ as shown in Figure 14. A step-relation is defined over this record that allows $\mathcal{A}$ to advance the state of the protocol as per the specifications of Pfitzmann and Waidner [3].

```
Record OriginalModel : Set := mkOriginalModel
          { A : Adversary
          ; Ps : list Party
          ; Fs : list FunctionBox }.
```

Figure 14: The `Record` representing the original model by Pfitzmann and Waidner in Coq.

4.2.1 An Example Step

To illustrate the idea behind the step-relation consider the example in Figure 16 of the step where a value on some outgoing buffer b_k^+ of a $\mathcal{Z}_i$ of some $\mathcal{P}_i$ to

the respective incoming buffer b_k^- of the $\mathcal{I}_i$ of that same $\mathcal{P}_i$ is clocked. Note that $\mathcal{I}_i$ will immediately produce a result and put it on an outgoing buffer b_l^+. This step is visualized in Figure 15 by the arrow labelled 1. This is achieved as

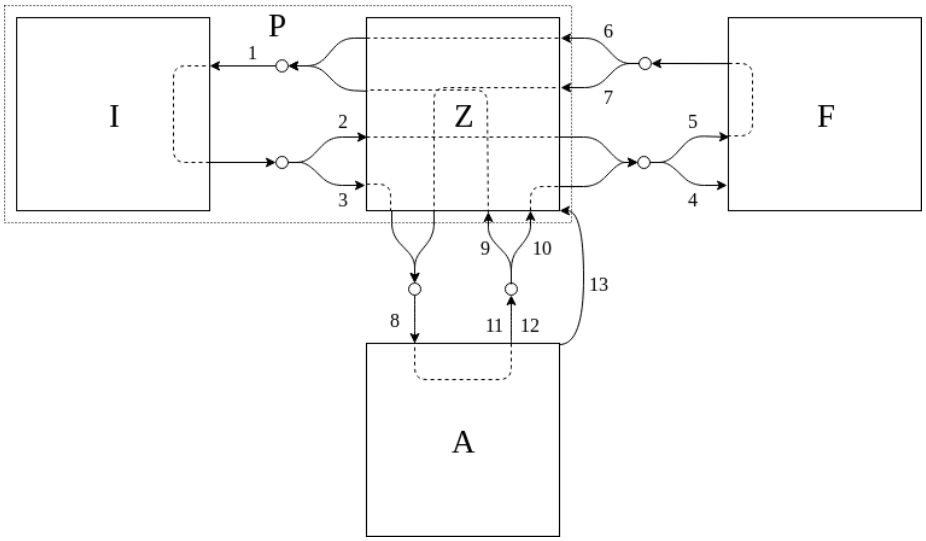

Figure 15: Each step-relation defined in the Coq model represented as an arrow in a diagram. Steps start and end at a circle (∘), following the direction of the arrow.

follows. The step is defined for any adversary `A` and any list of functionboxes `Fs`. This is because the details of the adversary and ideal functionalities are irrelevant when clocking a value from $\mathcal{Z}_i$ to $\mathcal{I}_i$. The step is also defined over any two lists of parties `PsNeck` and `PsTail`. These two lists represent all possible other parties present in the protocol. Combined with $\mathcal{P}_i$ constructed from $\mathcal{Z}_i$ and $\mathcal{I}_i$ these will form the list all parties in the protocol as:

```
(PsNeck ++ [mkParty I Z] ++ PsTail)
```

The step is also defined over all values `corrupted`, `bZincNeck`, `bZincTail`, `bZoutNeck`, and `bZoutTail`. These values compose the corruptor in the model, namely: a boolean value indicating if it is enabled (`corrupted`), a list of incoming buffers (`bZincNeck` and `bZincTail`), and a list of outgoing buffers (`bZoutNeck` and `bZoutTail`). The buffer that is clocked in the step will complete the list of outgoing buffers and the buffer to which $\mathcal{I}_i$ will output will complete the list of incoming buffers similar to the approach for the complete list of parties, as (for the list of incoming buffers):

```
(bZincNeck ++ [mkBuffer buffered released] ++ bZincTail)
```

Similarly, the step is also defined over all possible values that compose an interpreter in the model, namely: its state (`sIreceived` and `sIsend`), a list of

```
| STo_clock_Z_to_I :

forall (vinp vout : nat) (A : Adversary) (sIreceived sIsend : list nat) (bIincNeck bIincTail bIoutNeck
bIoutTail : list Buffer) (corrupted : bool) (bZincNeck bZincTail bZoutNeck bZoutTail : list Buffer)
(PsNeck PsTail : list Party) (Fs : list FunctionBox) (bsrc rsrc : list Message) (btar rtar : list Message)
(q : bool),

step_o

(mkOriginalModel
| A
| (PsNeck ++ [mkParty

   (mkInterpreter
   | (mkState sIreceived sIsend)
   | (bIincNeck ++ [mkBuffer ((mkMessage vinp q) ::  bsrc) rsrc] ++ bIincTail)
   | (bIoutNeck ++ [mkBuffer btar rtar] ++ bIoutTail)
   )

   (mkCorruptor
   | corrupted
   | (bZincNeck ++ [mkBuffer btar rtar] ++ bZincTail)
   | (bZoutNeck ++ [mkBuffer ((mkMessage vinp q) ::  bsrc) rsrc] ++ bZoutTail)
   )

| ] ++ PsTail)
| Fs
)

(mkOriginalModel
| A
| (PsNeck ++ [mkParty

   (mkInterpreter
   | (mkState
   || (if query then sIreceived else (vinp ::  sIreceived))
   || (vout ::  sIsend))
   | (bIincNeck ++ [mkBuffer bsrc (rsrc ++ [(mkMessage vinp q)])] ++ bIincTail)
   | (bIoutNeck ++ [mkBuffer (btar ++ [(mkMessage vout q)]) rtar] ++ bIoutTail)
   )

   (mkCorruptor
   | corrupted
   | (bZincNeck ++ [mkBuffer (btar ++ [(mkMessage vout q)]) rtar] ++ bZincTail)
   | (bZoutNeck ++ [mkBuffer bsrc (rsrc ++ [(mkMessage vinp q)])] ++ bZoutTail)
   )

| ] ++ PsTail)
| Fs
)
```

Figure 16: The Coq code for step 1 from Table 1. The red box contains the quantification of the step. The blue box contains the state of the protocol at time t. The green box contains the state of the protocol at time $t+1$ (i.e. after the step is applied). The yellow (specifying $\mathcal{I}_i$) and purple (specifying $\mathcal{Z}_i$) box are the components that change.

incoming buffers (`bIincNeck` and `bIincTail`), and a list of outgoing buffers (`bIoutNeck` and `bIoutTail`). For $\mathcal{I}_i$ the list of incoming buffers is completed by the buffer that is clocked in the step, whereas the list of outgoing buffers is completed by the buffer to which it outputs a value.

The step is also defined over all possible values for a source buffer and target buffer, namely, what values are currently buffered (`bsrc` and `btar` respectively), what values are already clocked (`rsrc` and `rtar` respectively). These values are used to construct the two buffers that complete the lists of buffers of $\mathcal{I}_i$ and $\mathcal{Z}_i$.

Finally, the step is defined for any input value (`vinp`), any output value (`vout`), and both query and non-query messages (`query`). This complete description of allowed values are then used to describe how a protocol in one state can be advanced to a next state. In general, this will be written as:

`step_o (mkOriginalModel` σ_t`) (mkOriginalModel` σ_{t+1}`)`

where σ_t specifies the state of the protocol must have before the step and σ_{t+1} the (possible) next state of the protocol.

In this particular step, the initial state requires that there is some value `vinp` on an outgoing buffer b_k^+ of $\mathcal{Z}_i$ to the respective incoming buffer b_k^- of $\mathcal{I}_i$ (named the source buffer). For this step, it does not matter if $\mathcal{Z}_i$ is enabled, which is captured by the fact that the boolean value `corrupted` can have any value.

When the step is taken, the value `vinp` is removed from the buffer and put on the released side of the source. $\mathcal{I}_i$ will compute a value and output it on some outgoing buffer b_l^+ to the respective incoming buffer b_l^- of $\mathcal{Z}_i$.

4.2.2 All steps

Every other step that is possible in Pfitzmann and Waidner model is defined similarly. Figure 15 provides a visual overview of the steps that have been defined in Coq, Table 1 provides an informal description of each step that was defined.

In Table 1 some steps have a precondition. Steps $\{2, 6, 13\}$ require that $\mathcal{Z}_i$ is disabled, and steps $\{3, 7, 8, 9, 10, 11, 12\}$ require that $\mathcal{Z}_i$ is enabled. This is modelled by explicitly setting the `corrupted` flag for $\mathcal{Z}_i$, as seen in Figure 16, to `false` and `true` respectively.

Furthermore, steps $\{4, 5\}$ have certain requirements on their incoming buffers. Step 4 (`STo_clock_Z_to_F`) has a precondition that less than $n-1$ buffers have a value clocked (where n is the number of inputs $\mathcal{F}_j$ expects). Similarly, step 5 (`STo_clock_Z_to_F_compute`) has a precondition that exactly $n-1$ buffers have a value clocked. In Coq this is achieved through a logical implications. On a high level this is written as:

```
precondition -> step_o (mkOriginalModel ...)(mkOriginalModel ...)
```

where the arrow denotes an implication. The precondition must result in the logical value `true` for the step to be applied to the protocol.

Table 1: All defined steps for Pfitzmann and Waidner model.

#	**Step name**	**Precondition(s)**	**Description**
1	`STo_clock_Z_to_I`	*None.*	Clock a message buffered in a b_k^+ of some $\mathcal{Z}_i$ to the respective b_k^- of $\mathcal{I}_i$. This results in $\mathcal{I}_i$ computing and outputting another message on a b_l^+ of $\mathcal{I}_i$ to the respective b_l^- of $\mathcal{Z}_i$.
2	`STo_clock_I_to_F`	This step can only be taken when $\mathcal{Z}_i$ is disabled.	Clock a value buffered in a b_k^+ of some $\mathcal{I}_i$ to the respective b_k^- of $\mathcal{Z}_i$ and put it on a b_l^+ of $\mathcal{Z}_i$ to the respective b_l^- of some $\mathcal{F}_j$.
3	`STo_clock_I_to_A`	This step can only be taken when $\mathcal{Z}_i$ is enabled.	Clock a message buffered in a b_k^+ of some $\mathcal{I}_i$ to the respective b_k^- of $\mathcal{Z}_i$ and put it on a b_l^+ of $\mathcal{Z}_i$ to the respective b_l^- of $\mathcal{A}$.
4	`STo_clock_Z_to_F`	This step can only be taken when less than $n-1$ incoming buffers b^- have a value clocked.	Clock a value buffered in a b_k^+ of some $\mathcal{Z}_i$ to the respective b_k^- of some $\mathcal{F}_j$.
5	`STo_clock_Z_to_F_compute`	This step can only be taken when exactly $n-1$ incoming buffers b^- have a value clocked.	Clock a value buffered in a b_k^+ of some $\mathcal{Z}_i$ to the respective b_k^- of some $\mathcal{F}_j$. This results in $\mathcal{F}_j$ computing and outputting another value on all b_l^+ of $\mathcal{F}_j$ to the respective b_l^- of all (connected) $\mathcal{Z}_i$.
6	`STo_clock_F_to_I`	This step can only be taken when $\mathcal{Z}_i$ is disabled.	Clock a value buffered in a b_k^+ of some $\mathcal{F}_j$ to the respective b_k^- of some $\mathcal{Z}_i$ and put it on a b_l^+ of $\mathcal{Z}_i$ to the respective b_l^- of $\mathcal{I}_i$.
7	`STo_clock_F_to_A`	This step can only be taken when $\mathcal{Z}_i$ is enabled.	Clock a value buffered in a b_k^+ of some $\mathcal{F}_j$ to the respective b_k^- of some $\mathcal{Z}_i$ and put it on a b_l^+ of $\mathcal{Z}_i$ to the respective b_l^- of $\mathcal{A}$.
8	`STo_clock_Z_to_A`	This step can only be taken when $\mathcal{Z}_i$ is enabled.	Clock a message buffered in a b_k^+ of some $\mathcal{Z}_i$ to the respective b_l^- of $\mathcal{A}$.
9	`STo_clock_A_to_I`	This step can only be taken when $\mathcal{Z}_i$ is enabled.	Clock a message buffered in a b_k^+ of $\mathcal{A}$ to the respective b_k^- of some $\mathcal{Z}_i$ and put it on a b_l^+ of $\mathcal{Z}_i$ to the respective b_l^- of $\mathcal{I}_i$.

Table 1: All defined steps for Pfitzmann and Waidner model.

#	Step name	Precondition(s)	Description
10	`STo_clock_A_to_F`	This step can only be taken when $\mathcal{Z}_i$ is enabled.	Clock a value buffered in a b_k^+ of $\mathcal{A}$ to the respective b_k^- of some $\mathcal{Z}_i$ and put it on a b_l^+ of $\mathcal{Z}_i$ to the respective b_l^- of some $\mathcal{F}_j$.
11	`STo_A_new_query`	This step can only be taken when $\mathcal{Z}_i$ is enabled.	Let $\mathcal{A}$ produce any new message containing a query and put it on a b_k^+ of $\mathcal{A}$ to the respective b_k^- of some $\mathcal{Z}_i$.
12	`STo_A_new_value`	This step can only be taken when $\mathcal{Z}_i$ is enabled.	Let $\mathcal{A}$ produce any new message containing a value and put it on a b_k^+ of $\mathcal{A}$ to the respective b_k^- of some $\mathcal{Z}_i$.
13	`STo_corrupt_party`	This step can only be taken when $\mathcal{Z}_i$ is disabled.	Let $\mathcal{A}$ corrupt a previously uncorrupt party $\mathcal{P}_i$.

4.3 Modelling the Modified Model

With Pfitzmann and Waidner model completed the next step is to model the modifications to the model by Laur et al. Using the `Record` for a composite adversary, as shown in Figure 17, a new `Record` representing the model is defined in Figure 17.

```
Record ModifiedModel : Set := mkModifiedModelModel
                { A : CompositeAdversary
                ; Ps : list Party
                ; Fs : list FunctionBox }.
```

Figure 17: The `Record` representing the modified model adopted by Laur et al. in Coq.

As for the original model, a step-relation is defined over this record that allows the adversary to advance the state of the protocol. Rather than going through another example, this section will highlight steps that are new and omitted w.r.t. the step-relation for the original model (see Appendix A for a complete overview of the step-relation in the modified model).

The first observation is that because of the top-level `Record` of the modified model is slightly different from the top-level `Record` of the original model, some of the steps need to be modified slightly so they can be applied. Two examples of affected steps are `STo_clock_I_to_A` and `STo_clock_Z_to_A`. This is a result

of embedding the real adversary within the composite adversary. These details are technical and are omitted here.

4.3.1 Communication Between $\mathcal{I}'_i$ and $\mathcal{A}'$

The modified model needs steps for the sub-components of the semi-simplistic adversary to communicate. Rather than defining a simple step that performs one communication cycle ($\mathcal{A}' \to \mathcal{I}'_i \to \mathcal{A}'$) at once, there are four separate steps for this communication as shown in Table 2. This design choice was made to facilitate proving the lemmas.

Table 2: Internal communication steps for a semi-simplistic adversary.

Step name	**Precondition(s)**	**Description**
`STm_clock_A'_to_I'`	*None.*	Clock a message buffered in a b_k^+ of $\mathcal{A}'$ to the respective b_k^- of $\mathcal{I}'_i$. This results in $\mathcal{I}'_i$ computing and outputting another message on a b_l^+ of $\mathcal{I}'_i$ to the respective b_l^- of $\mathcal{A}'$.
`STm_clock_I'_to_A'`	*None.*	Clock a message buffered in a b_k^+ of $\mathcal{I}'_i$ to the respective b_k^- of $\mathcal{A}'$.
`STm_A'_new_value`	*None.*	Let $\mathcal{A}$ produce any new message containing a value and put it on a b_k^+ of $\mathcal{A}$ to the respective b_k^- of $\mathcal{I}'_i$.
`STm_A'_new_query`	*None.*	Let $\mathcal{A}$ produce any new message containing a query and put it on a b_k^+ of $\mathcal{A}$ to the respective b_k^- of $\mathcal{I}'_i$.

4.3.2 Steps for Semi-Simplistic Adversaries

Recall the definition of a *Semi-Simplistic Adversary* given in Section 3.2. These requirements are encoded in the `Coq` model by adding preconditions or other restrictions to the steps. What follows is a description of how each requirement is encoded in the step-relation.

1. (*The adversary clocks outgoing buffers b^+ of parties only when all incoming buffers b^- to all corrupted parties are empty.*) This is modelled by adding a precondition on the step `STo_clock_Z_to_F` that for all parties $\mathcal{P}_i$ that are corrupted (i.e. the `is_corrupted` flag of $\mathcal{Z}_i$ is set to true, see Figure 11) all the incoming buffers (i.e. all buffers in the list of incoming buffers `z_incoming`, see Figure 11) are empty.

Because `STo_clock_Z_to_F` is the only step in which a party $\mathcal{P}_i$ releases a value into the world, this is the only step that needs to be modified for this requirement. Messages can still be queued for release by steps such as `STo_clock_I_to_F`, but as long as there is a non-empty incoming buffer b_k^- to some corrupted party, `STo_clock_Z_to_F` cannot be applied to the state of the protocol.

2. (*The adversary clock any incoming buffers b^- to an honest party only when*

all incoming buffers b^- to all corrupted parties are empty.) To model this requirement the `STo_clock_F_to_I` step is modified. Specifically, the precondition from the previous requirement is added to guarantee this requirement. Because `STo_clock_F_to_I` is the only step in which an incoming buffer from an honest party is clocked no other modifications are needed.

3. (*Upon receiving a message from a $\mathcal{F}_j$, the adversary immediately forwards it to the respective $\mathcal{I}_i$.*) This requirement requires a couple of changes to the `STo_clock_F_to_A` step. Note that it is not possible to replace it with `STo_clock_F_to_I` since the adversary will never get to see the value produced by $\mathcal{F}_j$. Hence, to force $\mathcal{A}$ to forward the message to $\mathcal{I}_i$ the step is modified so that it captures the entire path $\mathcal{F}_j \to \mathcal{Z}_i \to \mathcal{A} \to \mathcal{Z}_i \to \mathcal{I}_i$ at once.[7]

4. (*The adversary can send any message to a $\mathcal{F}_j$.*) Since $\mathcal{A}$ already had this capability no changes are needed.

5. (*The adversary can always send corruption instruction to $\mathcal{I}$ that do not alter the state of $\mathcal{I}$.*) This requirement affects the steps that move values between $\mathcal{A}$ and $\mathcal{I}_i$. It has been enforced through `STm_clock_Z_to_I` and `STm_clock_A_to_I` by only allowing the `corrupt` flag of messages (see Figure 5) to be `true` when the `query` flag is `true`.[8] In Coq this is written as:

```
(mkMessage v q (if q then true else false))
```

6. (*The adversary performs no other corrupt actions.*) Since $\mathcal{A}$ has no other corrupt actions to perform (after *requirement 5* is satisfied) no changes are needed.

4.4 Proof by Bisimulation

This section will discuss how the lemma (described in detail below) was proven using the Coq Proof Assistant. First, the lemma as it was formulated in Coq is explained in detail. Next, a part of the proof that illustrates the overall approach of the proof is explained in detail as well.

Bisimulation, in the context of the present work, means the simulation of both $\mathcal{A}$ in the original model by Pfitzmann and Waidner and the behaviourally restricted semi-simplistic adversary in the modified model at the same time. This to show that both can have the same influence over a protocol Π.

4.4.1 The Lemma

The lemma to be proven can informally be described as a semi-simplistic adversary in the modified model has equivalent adversarial capabilities to a be-

[7] Note that `STo_clock_F_to_I` does not need to be changed as it can only be used when $\mathcal{Z}_i$ is disabled.

[8] Alternatively, this could be modelled by modifying the step rule `STo_A_new_value` to put values on a buffer b^+ of $\mathcal{Z}_i$ in the direction of $\mathcal{F}_j$ immediately. However, this results in complications in the proof process.

haviourally unrestricted adversary in Pfitzmann and Waidner's model. In Coq this lemma must first be written down formally. Figure 18 shows how the lemma was written in Coq as an equivalence through bisimulation. The lemma consists of two preconditions, an antecedent which states the step in Pfitzmann and Waidner's model, and a consequent which states that one can take a step in the modified model and the state of $\mathcal{I}_i$ and $\mathcal{I}'_i$ are the same after the steps in each model.

But, before that, the lemma first states that it must hold for all possible $\mathcal{A}$s, $\mathcal{I}_i$s, (initial) $\mathcal{I}'_i$ states, $\mathcal{Z}_i$s, lists of $\mathcal{P}_i$s, and lists of $\mathcal{F}_j$s. Then, the first requirement is that the $\mathcal{Z}_i$ should be corrupt after the step has been taken (`oZ_a.(is_corrupted) = true`). The second requirement is that the state of $\mathcal{I}_i$ in the original model is equivalent to the state of $\mathcal{I}'_i$ in the modified model (`oI_b.(i_state) = mI's_b`[9]) before the step is taken.[10]

After the requirements, the lemma states that any step can be taken in the original model such that any component can change from state σ_t to state σ_{t+1}. This is expressed with the $\_b$ (before) and $\_a$ (after) suffixes on the component identifiers. It is explicitly modelled that there must be at least one party $\mathcal{P}_i$ in the list parties through `[mkParty oI_b oZ_b]` (`PsNeck` and `PsTail` may both be empty lists). This step is specified rather generously. For example, there are (currently) no steps that alter more than one party at the time but it is still allowed since `PsNeck_b` and `PsNeck_a` are allowed to be different.

What follows is an implication specifying a list of things that `exists` such that a step in the modified model can be taken. This means that, in the proof, one needs to specify the instances of these components so that one or more step for a semi-simplistic adversary are allowed by the model. Because $\mathcal{A}$ is behaviourally restricted in this setting, it is allowed to perform multiple actions through the `multistep_m` relation[11], which allows the prover to show equivalence after applying multiple steps to the protocol state. I.e. the modified model can advance from state σ_t to state σ_{t+k}.

Finally, it must also hold that the state of $\mathcal{I}_i$ after the step in the original model is the same as the state of $\mathcal{I}'_i$ after the step in the modified model (∧ `oI_a.(i_state) = mI'_a.(i_state).`).

4.4.2 Proof example

In general, the proof is carried out by proving, separately for every step, that there exist one or more steps for a semi-simplistic adversary to obtain `oI_a.(i_state) = mI'_a.(i_state).`. After doing some trivial rewriting and managing of knowledge as well as proving bisimulation of several steps (which are omitted here for brevity) one needs to proof bisimulation for the step `STo_clock_I_to_A` in the original model (line 3 in Figure 15). This subsection will briefly explain how

[9] There is a myriad of ways to specify this requirement, even implicit ones. The formulation here was chosen to make the requirement explicit.

[10] The requirements are formulated using implications (`->`) rather than conjunctions (∧) as this makes them easier to work with in Coq proofs.

[11] How this is obtained from the single step-relation is beyond the scope of the present work.

```
Theorem original_model_equivalent_to_modified_model :
(* For all components in the original model ...  *)
forall (oA_b oA_a :  Adversary) (oI_b oI_a :  Interpreter) (mI's_b :  State) (oZ_b
oZ_a :  Corruptor) (PsNeck_b PsTail_b PsNeck_a PsTail_a :  list Party) (oFs_b oFs_a
:  list FunctionBox),
(* if the corruptor is enabled ...  *)
|  oZ_a.(is_corrupted) = true
(* if the original and modified Interpreter have the same state initially ...  *)
-> oI_b.(i_state) = mI's_b
(* then for every step in the original model ...  *)
-> step_o (mkOriginalModel oA_b (PsNeck_b ++ [mkParty oI_b oZ_b] ++ PsTail_b) oFs_b)
|         (mkOriginalModel oA_a (PsNeck_a ++ [mkParty oI_a oZ_a] ++ PsTail_a) oFs_a)
(* there exists a step in them modified model ...  *)
-> exists (mI'_a :  Interpreter) (mI'inc_b mI'out_b :  list Buffer) (mA'_b mA'_a :
Adversary) (mI_b mI_a :  Interpreter) (mZ_b mZ_a :  Corruptor) (mFs_b mFs_a :  list
FunctionBox),
-> multistep_m
  (mkModifiedModel
  |  (mkCompositeAdversary mA'_b (mkInterpreter mI's_b mI'inc_b mI'out_b))
  |  (PsNeck_b ++ [mkParty mI_b mZ_b] ++ PsTail_b)
  |  mFs_b)
  (mkModifiedModel
  |  (mkCompositeAdversary mA'_a mI'_a)
  |  (PsNeck_a ++ [mkParty mI_a mZ_a] ++ PsTail_a)
  |  mFs_a)
(* such that the original and modified Interpreter end in the same state.  *)
∧ oI_a.(i_state) = mI'_a.(i_state).
Proof.  ...  Qed.
```

Figure 18: The lemma to be proven for Laur et al. formulated in Coq. The proof itself is omitted here, the proof of a single case can be found in Figure 19. The red boxes contain universal and existential quantification. The green box contains the pre-requirements for the lemma. The blue boxes contain the step descriptions. And finally, the yellow box contains the requirement on the state of $\mathcal{I}_i$ and $\mathcal{I}'_i$.

this can be done. All other cases can be proven in a similar fashion, although the proof of some other cases is more involved than the one discussed here.

As shown in Figure 19 one starts to prove a case by stating the instances of the components for the step in the modified model. First, we specify that `mI'_a` is instantiated by `oI_a`. This is the simplest way to guarantee the states of $\mathcal{I}_i$ and $\mathcal{I}'_i$ are equal after the steps. Next, we specify that we want to use the same adversary (`oA_b` and `oA_a`), interpreter (`oI_b` and `oI_a`), corruptor (`oZ_b` and `oZ_a`), and list of function boxes (`oFs_b` and `oFs_a`).

For the list of incoming buffers (b^-) and outgoing buffers (b^+) of $\mathcal{I}'_i$ (the third `exists`) we need to be more specific. The list of incoming buffers comes straight from the list of incoming buffers for `oI_b` as `bIinc`. However, since there is a value on the outgoing buffer of `oI_a` Coq does no allow one to say `exists bIout` as well. Instead, one needs to construct this list from its component parts, namely `bIoutNeck`; the buffer of interest consisting of `bsrc`, `rsrc`, and the message with value `v`; and `bIoutTail`.

```
- (* STo_clock_I_to_A *)
```

```
(* Specify the state in the modified model.  *)
exists oI_a
exists oA_b, oA_a.
exists bIinc, (bIoutNeck ++
|                [mkBuffer bsrc (rsrc ++ [(mkMessage v q false)])] ++ bIoutTail).
exists oI_b, oI_a.
exists oZ_b, oZ_a.
exists oFs_b, oFs_a.
```

```
(* Proof the goal.  *)
split.
- rewrite <- H, (* Hypothesis about the Adversary (after).  *)
|          <- H0, (* Hypothesis about the Adversary (before).  *)
|          H5, (* Hypothesis about the list of functions.  *)
|          HoI_b, HoI_a, HoZ_b, HoZ_a, HPsNeck_b, HPsNeck_a, HPsTail_b, HPsTail_a.
|  apply STm_clock_I_to_A.
- reflexivity.
```

Figure 19: An example of how a single step can be proven in Coq. This is the case for the `STo_clock_I_to_A` step for the lemma in Figure 18. The red box describes the concretization of the proof goal and the blue box describes the proof steps.

Once the components with which the step is to be taken are concretized, the state of the goal has evolved from the goal in Figure 20a to the goal in Figure 20b. With the goal finalized the goal can be proven. The logical and ($\wedge$) in the goal can be `split` into two separate goals. Then, first, one needs to prove that `multistep_m` σ_t σ_{t+k} is a possible step. And, second, one needs to prove that

the states of $\mathcal{I}_i$ and $\mathcal{I}'_i$ are the same after the respective steps (`oI_a.(i_state) = mI'_a.(i_state)`).

To prove the first goal, one first `rewrite`s the goal with several hypotheses. These hypotheses are generated by Coq from the step in the original model. For example, hypothesis `H0` and `H` in this example contain the complete state of $\mathcal{A}$ before and after (respectively) the step as taken. After performing the rewrites, the goal evolves to what is shown in Figure 20c. Now, the goal describes exactly the step `STm_clock_I_to_A`, hence we apply that step to the goal state and find that it is now proven.[12]

Next, one needs to prove that the states of $\mathcal{I}_i$ and $\mathcal{I}'_i$ are the same after the respective steps, as shown in Figure 20d. Since the first exists allows one to specify what $\mathcal{I}'_i$ is after the step is taken, and we specified that it is `oI_a`, it is trivial to show that `oI_a.(i_state) = oI_a.(i_state)`. In Coq one uses the `reflexivity` tactic for this.

Hence, it is now proven that whenever $\mathcal{A}$ in the original model performs the step `STo_clock_I_to_A`, a semi-simplistic adversary in the modified model can perform the step `STm_clock_I_to_A` to keep $\mathcal{I}_i$ and $\mathcal{I}'_i$ synchronized. To proof the full lemma, one needs to show a proof like this exists for every step defined in the original model (see Section 4.2.2).

5 Discussion

Design choices The general approach in the design process was to define a step-relation over the entire protocol state. This was suggested by K. Apinis and is inspired by the small-step semantics of simply-typed lambda calculus [4, 5]. This is not the only valid modelling strategy in Coq. Various programming constructs are available in Coq and other approaches might be (more) suitable. For example, one could define advancing the protocol using *inductive definitions* and *fixpoint operators* instead.

The final product described here could have small design changes as well. For example, the step-relations for the two models have quite some similarity (as evidenced by Appendix A). This is a result of the models (and adversaries) being defined as a separate `Record` between the two. Using a different design it might be possible to reduce the amount of replication.

Drawbacks The current modelling in Coq has some notable drawbacks. First, there is no guarantee that the step-relations that have been defined accurately capture all possible steps in the respective models. On the plus, side it is relatively easy to add new steps to the step-relations and update the proofs accordingly. Second, the step-relations are rather specific and prone to errors. For example, each buffer is modelled twice (one incoming and one outgoing buffer) and Coq provides no guarantee that a connected pair of buffers (b_k^-, b_k^+) stay

[12] This is simplified to hide details about `multistep`, which is beyond the scope of the present work.

```
exists (mI'_a :  Interpreter) (mA'_b mA'_a :  Adversary) (mI'inc_b mI'out_b :  list Buffer)
(mI_b mI_a :  Interpreter) (mZ_b mZ_a :  Corruptor) (mFs_b mFs_a :  list FunctionBox),
multistep_m
{ A := { A' := mA'_b;
|        I' := { i_state := sI; i_incoming := mI'inc_b; i_outgoing := mI'out_b } };
| Ps := PsNeck_b ++ [{ I := mI_b; Z := mZ_b }] ++ PsTail_b;
| Fs := mFs_b }
{ A := { A' := mA'_a; I' := mI'_a };
| Ps := PsNeck_a ++ [{ I := mI_a; Z := mZ_a }] ++ PsTail_a;
| Fs := mFs_a }
∧ oI_a.(i_state) = mI'_a.(i_state)
```

(a) The initial goal state for Figure 19.

```
multistep_m
{ A := { A' := oA_b;
|        I' := { i_state := sI; i_incoming := bIinc; i_outgoing := (bIoutNeck ++
|        [mkBuffer ilsrc bsrc (rsrc ++ [(mkMessage v q false)]) lsrc] ++ bIoutTail) } };
| Ps := PsNeck_b ++ [{ I := oI_b; Z := oZ_b }] ++ PsTail_b;
| Fs := oFs_b }
{ A := { A' := oA_a; I' := oI_a };
| Ps := PsNeck_a ++ [{ I := oI_a; Z := oZ_a }] ++ PsTail_a;
| Fs := oFs_a }
∧ oI_a.(i_state) = oI_a.(i_state)
```

(b) The goal state after concretizing `exists`.

```
multistep_m
{ A := { A' := oA_b; I' := oI_a };
| Ps := PsNeck_a ++ [{ I := oI_b; Z := oZ_b }] ++ PsTail_a;
| Fs := oFs_a }
{ A := { A' := oA_a; I' := oI_a };
| Ps := PsNeck_a ++ [{ I := oI_a; Z := oZ_a }] ++ PsTail_a;
| Fs := oFs_a }
```

(c) The goal state after rewriting.

```
∧ oI_a.(i_state) = oI_a.(i_state)
```

(d) The last goal state for the case in Figure 19.

Figure 20: The four intermediate goal states in Coq for proving the bisimulation of `STo_clock_I_to_A`.

synchronized. This limitation could potentially be resolved through a lemma that guarantees that any pair of buffers (b_k^-, b_k^+) always contain the same values.

Future work The final product of the present work is not a 100% accurate representation of Pfitzmann and Waidner's model. The shortcomings include:

- There is no step-relation in which an interpreter $\mathcal{I}_i$ halts and no longer responds to any input. This could be modelled in two ways. The first possible implementation is a new boolean field in the $\mathcal{I}_i$ `Record` that is `true` by default and can be changed, with a new step-relation `ST_clock_Z_to_I_halt`, to `false`. When the new field's value is `false` it is either impossible to clock to $\mathcal{I}_i$ or upon receiving a new message it will not output a value. The second possible implementation is to have a new step-relation, `ST_clock_Z_to_I_halt`, where the state of the protocol collapses into something unusable (e.g. the list of parties becomes an empty list). The latter is easier to add to the current Coq model.
- The modified model representing the model of Laur et al. currently has only a single copy of an $\mathcal{I}'$ inside the composite adversary. A more accurate design would be to have a separate $\mathcal{I}'_i$ for every party $\mathcal{P}_i$. Implementing this is somewhat challenging as it requires a one-to-one mapping between $\mathcal{I}'_i$ and $\mathcal{P}_i$. This could be achieved by using a list of $\mathcal{I}'$ and requiring that the index of $\mathcal{I}'_i$ in this list corresponds to the index of $\mathcal{P}_i$ in the list of parties.

Furthermore, Laur's work doesn't stop where the present work stops. The Coq model build can be utilized to analyze other aspects of Laur's work as well as prove other lemmas formally.

6 Conclusion

The results of the present work show that the particular lemma discussed does indeed hold. I.e. an adversary under the behaviour restrictions of a semi-simplistic adversary (see Section 3.2) in Pfitzmann and Waidner's model has equivalent adversarial capabilities as a behaviourally unrestricted adversary (w.r.t. the state of $\mathcal{I}_i$ and $\mathcal{I}'_i$). However, this results rests on the assumption that the Pfitzmann and Waidner's work, as well as Laur's work, was correctly and fully modelled in Coq.

Acknowledgement

I would like S. Laur for their guidance and K. Apinis for answering questions regarding the Coq Proof Assistant.

References

[1] Welcome! — the coq proof assistant. https://coq.inria.fr/. Accessed: 2019-11-11.

[2] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The coq proof assistant reference manual: Version 6.1. 1997.

[3] Birgit Pfitzmann and Michael Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, pages 184–200. IEEE, 2000.

[4] Benjamin C Pierce and C Benjamin. *Types and programming languages*. MIT press, 2002.

[5] Benjamin C Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. Software foundations. *Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html*, 2010.

A Description of All Steps in the Modified Model

A.1 Modified steps

In Table 3 the first column gives the number of the requirement of a semi-simplistic adversary that the modified step aims to satisfy.

Table 3: All modified steps for the modified model.

#	Step name	Precondition(s)	Description
1	`STm_clock_Z_to_F`	This step can only be taken when all $\mathcal{P}_i$ that are corrupted have all incoming buffers b^- empty.	Clock a value buffered in a b_k^+ of some $\mathcal{Z}_i$ to the respective b_k^- of some $\mathcal{F}_j$.
1	`STm_clock_Z_to_F_compute`	This step can only be taken when all $\mathcal{P}_i$ that are corrupted have all incoming buffers b^- empty.	Clock a value buffered in a b_k^+ of some $\mathcal{Z}_i$ to the respective b_k^- of some $\mathcal{F}_j$. This results in $\mathcal{F}_j$ computing and outputting another value on all b^+ of $\mathcal{F}_j$ to the respective b^- of all (connected) $\mathcal{Z}_i$.
2	`STm_clock_F_to_I`	This step can only be taken when $\mathcal{Z}_i$ is disabled and all $\mathcal{P}_i$ that are corrupted ave all incoming buffers b^- empty.	Clocks a value buffered in a b_k^+ of some $\mathcal{F}_j$ to the respective b_k^- of some $\mathcal{Z}_i$ and put it on a b_l^+ of $\mathcal{Z}_i$ to the respective b_l^- of $\mathcal{I}_i$.
3	`STm_clock_F_to_I_corrupt`	This step can only be taken when $\mathcal{Z}_i$ is enabled.	Originally named `STm_clock_F_to_A` this step clocks a value buffered on a b_k^+ of some $\mathcal{F}_j$ to the respective b_k^- of $\mathcal{Z}_i$. Next the value is put on a b_l^+ of $\mathcal{Z}_i$ to the respective b_l^- of $\mathcal{A}$ and clocked so that $\mathcal{A}$ receives it. Next, the value is clocked through a b_m^+ of $\mathcal{A}$ to the respective b_m^- of $\mathcal{Z}_i$. Finally, the value is put on a b_n^+ of $\mathcal{Z}_i$ to the respective b_n^- of $\mathcal{I}_i$. This step can still only be taken when $\mathcal{Z}_i$ is enabled.
5	`STm_clock_Z_to_I`	This step can only be taken when the value of the `corrupt` flag of messages (see Figure 5) equals the value of the `query` flag.	Clock a message buffered in a b_k^+ of some $\mathcal{Z}_i$ to the respective b_k^- of $\mathcal{I}_i$. This results in $\mathcal{I}_i$ computing and outputting another message on a b_l^+ of $\mathcal{I}_i$ to the respective b_l^- of $\mathcal{Z}_i$.

Table 3: All modified steps for the modified model.

#	Step name	Precondition(s)	Description
5	`STm_clock_A_to_I`	This step can only be taken when $\mathcal{Z}_i$ is enabled and the value of the `corrupt` flag of messages (see Figure 5) equals the value of the `query` flag.	Clock a message buffered in a b_k^+ of $\mathcal{A}$ to the respective b_k^- of some $\mathcal{Z}_i$ and put it on a b_l^+ of $\mathcal{Z}_i$ to the respective b_l^- of $\mathcal{I}_i$.

A.2 New steps

Table 4: Internal communication steps for a semi-simplistic adversary.

Step name	Precondition(s)	Description
`STm_clock_A'_to_I'`	*None.*	Clock a message buffered in a b_k^+ of $\mathcal{A}'$ to the respective b_k^- of $\mathcal{I}_i'$. This results in $\mathcal{I}_i'$ computing and outputting another message on a b_l^+ of $\mathcal{I}_i'$ to the respective b_l^- of $\mathcal{A}'$.
`STm_clock_I'_to_A'`	*None.*	Clock a message buffered in a b_k^+ of $\mathcal{I}_i'$ to the respective b_k^- of $\mathcal{A}'$.
`STm_A'_new_value`	*None.*	Let $\mathcal{A}$ produce any new message containing a value and put it on a b_k^+ of $\mathcal{A}$ to the respective b_k^- of $\mathcal{I}_i'$.
`STm_A'_new_query`	*None.*	Let $\mathcal{A}$ produce any new message containing a query and put it on a b_k^+ of $\mathcal{A}$ to the respective b_k^- of $\mathcal{I}_i'$.

A.3 Unmodified steps

Table 5: All unmodified steps for the modified model.

Step name	Precondition(s)	Description
`STm_clock_I_to_F`	This step can only be taken when $\mathcal{Z}_i$ is disabled.	Clock a value buffered in a b_k^+ of some $\mathcal{I}_i$ to the respective b_k^- of $\mathcal{Z}_i$ and put it on a b_l^+ of $\mathcal{Z}_i$ to the respective b_l^- of some $\mathcal{F}_j$.
`STm_clock_I_to_A`	This step can only be taken when $\mathcal{Z}_i$ is enabled.	Clock a message buffered in a b_k^+ of some $\mathcal{I}_i$ to the respective b_k^- of $\mathcal{Z}_i$ and put it on a b_l^+ of $\mathcal{Z}_i$ to the respective b_l^- of $\mathcal{A}$.
`STm_clock_F_to_A`	This step can only be taken when $\mathcal{Z}_i$ is enabled.	Clock a value buffered in a b_k^+ of some $\mathcal{F}_j$ to the respective b_k^- of some $\mathcal{Z}_i$ and put it on a b_l^+ of $\mathcal{Z}_i$ to the respective b_l^- of $\mathcal{A}$.

Table 5: All unmodified steps for the modified model.

Step name	**Precondition(s)**	**Description**
`STm_clock_Z_to_A`	This step can only be taken when $\mathcal{Z}_i$ is enabled.	Clock a message buffered in a b_k^+ of some $\mathcal{Z}_i$ to the respective b_l^- of $\mathcal{A}$.
`STm_clock_A_to_F`	This step can only be taken when $\mathcal{Z}_i$ is enabled.	Clock a value buffered in a b_k^+ of $\mathcal{A}$ to the respective b_k^- of some $\mathcal{Z}_i$ and put it on a b_l^+ of $\mathcal{Z}_i$ to the respective b_l^- of some $\mathcal{F}_j$.
`STm_A_new_query`	This step can only be taken when $\mathcal{Z}_i$ is enabled.	Let $\mathcal{A}$ produce any new message containing a query and put it on a b_k^+ of $\mathcal{A}$ to the respective b_k^- of some $\mathcal{Z}_i$.
`STm_A_new_value`	This step can only be taken when $\mathcal{Z}_i$ is enabled.	Let $\mathcal{A}$ produce any new message containing a value and put it on a b_k^+ of $\mathcal{A}$ to the respective b_k^- of some $\mathcal{Z}_i$.
`STm_corrupt_party`	This step can only be taken when $\mathcal{Z}_i$ is disabled.	Let $\mathcal{A}$ corrupt a previously uncorrupt party $\mathcal{P}_i$.

B Abbreviations Used in Coq Code

Table 6: All abbreviations used in Coq code that are present in this paper.

Abbriviation	Full	Description
mkAdversary	Make Adversary	Constructor for the $\mathcal{A}$ Record.
mkBuffer	Make Buffer	Constructor for the buffer Record.
mkCompositeAdversary	Make Composite Adversary	Constructor for the composite $\mathcal{A}$ Record.
mkCorruptor	Make Corruptor	Constructor for the $\mathcal{Z}_i$ Record.
mkFunctionBox	Make Function Box	Constructor for the $\mathcal{F}_j$ Record.
mkInterpreter	Make Interpreter	Constructor for the $\mathcal{I}_i$ Record.
mkMessage	Make Message	Constructor for the message Record.
mkParty	Make Party	Constructor for the $\mathcal{P}_i$ Record.
mkState	Make State	Constructor for the state Record.
a_state	Adversary State	The state of the $\mathcal{A}$ Record.
a_incoming	Adversary Incoming Buffers	The list of incoming of the $\mathcal{A}$ Record.
a_outgoing	Adversary Outgoing Buffers	The list of outgoing of the $\mathcal{A}$ Record.
f_state	Function Box State	The state of the $\mathcal{F}_j$ Record.
f_incoming	Function Box Incoming Buffers	The list of incoming of the $\mathcal{F}_j$ Record.
f_outgoing	Function Box Outgoing Buffers	The list of outgoing of the $\mathcal{F}_j$ Record.
i_state	Interpreter State	The state of the $\mathcal{I}_i$ Record.
i_incoming	Interpreter Incoming Buffers	The list of incoming of the $\mathcal{I}_i$ Record.
i_outgoing	Interpreter Outgoing Buffers	The list of outgoing of the $\mathcal{I}_i$ Record.
z_incoming	Corruptor Incoming Buffers	The list of incoming of the $\mathcal{Z}_i$ Record.
z_outgoing	Corruptor Outgoing Buffers	The list of outgoing of the $\mathcal{Z}_i$ Record.
Ps	Parties	A list of parties $\mathcal{P}_i$.
PsNeck	Parties Neck	The neck (first $l < n$ elemens) of a list of parties $\mathcal{P}_i$ (of length n).
PsNeck	Parties Tail	The tail (last $l < n$ elemens) of a list of parties $\mathcal{P}_i$ (of length n).
Fs	Parties	A list of function boxes $\mathcal{F}_j$.
FsNeck	Function Boxes Neck	The neck (first $l < n$ elemens) of a list of function boxes $\mathcal{F}_j$ (of length n).
FsNeck	Function Boxes Tail	The tail (last $l < n$ elemens) of a list of function boxes $\mathcal{F}_j$ (of length n).
bZincNeck	Buffers $\mathcal{Z}_i$ Incoming Neck	The neck (first $l < n$ elemens) of the list of incoming buffers b^- of a $\mathcal{Z}_i$ (of length n).
bZincTail	Buffers $\mathcal{Z}_i$ Incoming Tail	The tail (last $l < n$ elemens) of the list of incoming buffers b^- of a $\mathcal{Z}_i$ (of length n).

Table 6: All abbreviations used in Coq code that are present in this paper.

Abbriviation	**Full**	**Description**
`bZoutNeck`	Buffers $\mathcal{Z}_i$ Outgoing Neck	The neck (first $l < n$ elemens) of the list of outgoing buffers b^+ of a $\mathcal{Z}_i$ (of length n).
`bZoutTail`	Buffers $\mathcal{Z}_i$ Outgoing Tail	The tail (last $l < n$ elemens) of the list of outgoing buffers b^+ of a $\mathcal{Z}_i$ (of length n).
`bIinc`	Buffers $\mathcal{I}_i$ Incoming	The list of incoming buffers b^- of an $\mathcal{I}_i$ (of length n).
`bIincNeck`	Buffers $\mathcal{I}_i$ Incoming Neck	The neck (first $l < n$ elemens) of the list of incoming buffers b^- of an $\mathcal{I}_i$ (of length n).
`bIincTail`	Buffers $\mathcal{I}_i$ Incoming Tail	The tail (last $l < n$ elemens) of the list of incoming buffers b^- of an $\mathcal{I}_i$ (of length n).
`bIoutNeck`	Buffers $\mathcal{I}_i$ Outgoing Neck	The neck (first $l < n$ elemens) of the list of outgoing buffers b^+ of an $\mathcal{I}_i$ (of length n).
`bIoutTail`	Buffers $\mathcal{I}_i$ Outgoing Tail	The tail (last $l < n$ elemens) of the list of outgoing buffers b^+ of an $\mathcal{I}_i$ (of length n).
`sIreceived`	State $\mathcal{I}_i$ Received	The list of elements received by an $\mathcal{I}_i$.
`sISend`	State $\mathcal{I}_i$ Send	The list of elements send by an $\mathcal{I}_i$.
`bsrc`	Buffered source	The buffered values from which a value is coming in a step.
`rsrc`	Received source	The received values from which a value is coming in a step.
`btar`	Buffered target	The buffered values to which a value is going in a step.
`rtar`	Received target	The received values to which a value is going in a step.
`vinp`	Value input	The value that is the input in a step.
`vout`	Value output	The value that is the output in a step.
`oA_b`	Original $\mathcal{A}$ before	The instance of $\mathcal{A}$ in the original model before the step was applied.
`oA_a`	Original $\mathcal{A}$ after	The instance of $\mathcal{A}$ in the original model after the step was applied.
`oI_b`	Original $\mathcal{I}_i$ before	The instance of $\mathcal{I}_i$ in the original model before the step was applied.
`oI_A`	Original $\mathcal{I}_i$ after	The instance of $\mathcal{I}_i$ in the original model after the step was applied.

Table 6: All abbreviations used in Coq code that are present in this paper.

Abbriviation	**Full**	**Description**
`oZ_b`	Original $\mathcal{Z}_i$ before	The instance of $\mathcal{Z}_i$ in the original model before the step was applied.
`oZ_A`	Original $\mathcal{Z}_i$ after	The instance of $\mathcal{Z}_i$ in the original model after the step was applied.
`oFs_b`	Original list of $\mathcal{F}_j$ before	The instance of the list of $\mathcal{F}_j$ in the original model before the step was applied.
`oFs_A`	Original list of $\mathcal{F}_j$ after	The instance of the list of $\mathcal{Z}_i$ in the original model after the step was applied.
`H`	Hypothesis	A hypothesis.
`Hx`	Hypothesis x	A numbered hypothesis (e.g. `H0`).
`Hname`	Hypothesis *name*	A named hypothesis (e.g. `HoI_b`).