Information Security Research Institute

Modelling a cryptographic protocol with the purpose of formal verification in Isabelle

Version 1.0

Johanna Maria Kirss

D-2-457 / 2022

Mailing address:
Cybernetica AS
Mäealuse 2/1
12618 Tallinn
Estonia

Modelling a cryptographic protocol with the purpose of formal verification in Isabelle

Johanna Maria Kirss

Version 1.0

Abstract

This report reviews a research project in modelling a cryptographic protocol in Isabelle, a proof assistant. It details its premises, theoretical framework, implementation and reflects on the outcomes of the project.

Contents

1 Introduction

The aim of this document is to report on a formal verification project done by Johanna Maria Kirss, Sven Laur and with the aid of Dominique Unruh. The project consisted of modelling of a cryptographic protocol in the proof assistant Isabelle in order to formally verify a result about the protocol. While originally the goal was to prove a theorem detailed in the appendix, the project was later adjusted to just the modelling of the protocol.

In section 2, we introduce some important preliminaries like formal verification and the proof assistant Isabelle, secure multiparty computation, and an outline of a paper by Pille Pullonen-Raudvere and Sven Laur that this project is based on. Section 3 of the report introduces the theoretical framework of the project. That is, it surveys the reactive simulatability formalism and shows how secure multiparty computation can be modelled within it – what kinds of participants there are, and what their possible actions are. Section 4 explains the result to be proven as well as the Isabelle code. It introduces some auxiliary work done. The report closes with a concluding note on the current state of the research and further directions. The appendix surveys a larger theorem the project originally meant to tackle, before the scope of the project was adjusted.

2 Preliminaries

2.1 Secure multiparty computation

Secure multiparty computation (MPC) is a general term for the process of finding results from multiple inputs while keeping those inputs private. It can be a simple procedure like determining the the richer person of two while keeping their incomes secret [Yao82], or a complex one like collecting browser user statistics ([CGB17]). The range of use cases for MPC is impressive, including e.g. determining illegal passengers of a flight (by securely finding the intersection between a carrier's ticket holders and a local authority's no-fly list) ([FNP04]), and obscuring the memory usage patterns of a client in public storage ([Fle16]).

2.2 Foundations of programmable secure multiparty computation

This project is a part of a larger body of work being done by Pille Pullonen-Raudvere and Sven Laur. The relevant paper, whose working title is "Foundations of programmable secure multiparty computation", builds an abstract framework to describe MPC protocols. The paper is not publicly available yet, but in extremely brief terms, it is a series of theorems transforming a general MPC protocol to an equivalent, but simpler and more abstract protocol. It achieves an abstract model of MPC execution. That model is most suited to the Sharemind architecture but also fits other multiparty solutions.

2.3 The goal of this project

At the outset, the aim of this project was to verify the proof of one of such intermediate theorems in that paper. That is, the relevant theorem would have been about the equivalence of two protocols when considering the distribution of their final outputs. It does have a pen-and-paper proof, however, since it is about the properties of two complex distributed systems, a formal proof would have greatly reduced the uncertainty of whether the proof's correctness.

So, in light of that, the original structure of the project was as follows. First, it would be necessary to describe the two different models of multiparty computation in Isabelle. Once we have set up both models in functional language, the equivalence of the models would be proven by a series of formal statements. The formal reasoning would follow the pen-and-paper proof. The crucial part of the proof would have been a relation in between system states similar to a bisimulation. Its purpose is to show that the states, in some precise sense, operate in "lockstep" and provide the same outputs for related states.

However, the modelling of the protocol proved a much more complex task than originally thought (the issues are touched on in the conclusion), it became clear that both modelling and proving that result in one semester is infeasible. In response, the project became an exercise in modelling a complex real-life distributed system in functional language, rather than an in formally verifying any theorem. Of course, the modelling is affected by the choice of theorem, since a model is informed by what is relevant with respect to some purpose. To that end, we did choose a small statement to prove to aim for. This result is explained at the end of 4.

3 Theoretical framework

3.1 Formal proofs and Isabelle

For a more thorough guide to formal methods, see the reading list at [FM].

Consider a mathematical theorem. A theorem usually has a proof, written by one or several mathematicians – call this a **pen-and-paper proof**. While pen-and-paper proofs are the norm, convoluted or massive arguments may be too slippery, sometimes even impossible, for a human mind to treat with rigor. This can happen with complex mathematical proofs, but also when one tries to prove properties about complex systems – for example, distributed computing systems.

In these cases, proof assistants and formal verification may be a valuable addition in a scientist's toolbox. Where a human would get disoriented or accidentally neglect parts of the argument, a computer can reason methodically about every possible case.

So, in addition to the result and its pen-and-paper proof, a mathematical argument can also be built with a computer. This can be called a formal proof. To build formal proofs, different **proof assistants** – tailored programming languages – have been created. Its user may then transcribe their result into the proof assistant, and using suitable methods of the assistant, build a formal proof of the result.

Isabelle is one such proof assistant. It is a functional programming language, and it is tailored specifically to expressing statements in formal language, and also building their proofs with logical calculus. Here is an example of a theorem from the Isabelle tutorial [NPW10].

```
datatype 'a list = Nil ("[]")
| Cons 'a "'a list" (infixr "#" 65)

(* This is the append function: *)
primrec app :: "'a list => 'a list => 'a list" (infixr "@" 65)
where
"[] @ ys = ys" |
"(x # xs) @ ys = x # (xs @ ys)"

primrec rev :: "'a list => 'a list" where
"rev [] = []" |
"rev (x # xs) = (rev xs) @ (x # [])"

theorem rev_rev [simp]: "rev(rev xs) = xs"
```

In this example, a list datatype is first defined inductively. Then, a concatenating function app is defined, as well as a function rev that reverses the list. Finally it is stated that reversing a list twice yields the original result.

As can somewhat be seen above, mathematical results are not the only things one can reason about in Isabelle. One can also prove statements about data structures (as shown in the example), or even model a larger structure (for example, a network protocol) as a function or datatype, and prove properties of that structure. The latter is what this project set out to achieve.

3.2 Reactive simulatability

In the paper, secure multiparty computation is modelled as an asynchronous distributed system, and thus uses a conceptual and visual framework based on reactive simulatability (RSIM). For a detailed treatment of the reactive simulatability, see [BPW04].

In the RSIM framework, distributed systems are made up of **machines**, modelling parties or system components. The machines are capable of delivering and receiving asynchronous messages, and this is modelled using **buffers**. Messages can be written into buffers, and at a later time, the recipient can clock (process) said message. A finite set of machines and buffers is called a **collection**.

The buffers are connected to the machines using **ports** – ports can be for input or output, and some can also send clocking signals. Interestingly, a port is called a free port if it belongs to a machine in the collection, but is not connected to any other machine or buffer in the system. Between a machine and a port is a **connector**; a free connector is attached to a buffer but to no machine in the system.

If two collections can be merged by joining free port and free connector pairs in a way that respects the input/output type of the ports, then the collections are said to have **matching interfaces**. For two collections $\mathcal{C}_1, \mathcal{C}_2$ with matching interfaces, the result of their merging is denoted $\mathcal{C}_1\langle\mathcal{C}_2\rangle$. If more than two collections are joined in this manner, it can be denoted as $\mathcal{C}_1\langle\mathcal{C}_2, \mathcal{C}_3\rangle$ or $\mathcal{C}_1\langle\mathcal{C}_2\langle\mathcal{C}_3\rangle\rangle$. Finally, if a collection has no free ports or connectors, it is called a **closed** collection.

3.3 Execution model

In this project, the model is a closed collection of three collections: Π, the collection modelling the actual protocol, Env, an environment collection with a minimal role in this project, and A, the adversary. The collection is denoted as $\mathsf{Env}\langle\Pi, \mathsf{A}\rangle$.

The protocol Π has two specific kinds of machines – protocol parties written as $\mathcal{P}_i$ and functionalities written as $\mathcal{F}_k$. A protocol party really consists of two machines – an interpreter written as $\mathcal{I}_i$, and a corruption module written as $\mathcal{Z}_i$.

Both **protocol parties** and **functionalities** have sets of public and private parameters, as well as incoming and outgoing buffers, and both types of machines keep some sort of state that they use to process messages. An important difference between the machines is that a protocol party can be corrupted.

A party's **interpreter** $\mathcal{I}_i$ processes incoming messages according to the state it keeps of the party. That is, an interpreter keeps track of the party's code, program counter (where in the code the party is with its execution), buffers and stored values, and when there is an incoming message, it processes the message according to those and its internal semantics.

The **corruption module** $\mathcal{Z}_i$, however, acts differently depending on whether the party is in the honest or corrupted state. If it is honest, it forwards received messages to the interpreter for processing, but if corrupted, it sends them to the adversary without calling the interpreter.

A **functionality** $\mathcal{F}_k$ has a complex inner architecture that was less relevant to detail in the implementation. It consists of three modules – the sharing, reconstruction and computation modules – which communicate with the parties, reconstruct information and perform computations. When processing a message, it uses those three modules instead of a single interpreter, but the execution of that was left unspecified in this project.

The **environment** Env was also left quite unspecified as its function was not relevant to either the original or later result we aimed to prove.

3.4 The adversary

The adversary, in this model, can corrupt parties, peek into buffers or clock messages to their recipients, send new messages from corrupted parties, query functionalities and the environment. We describe here what the adversary does during those actions, and then in section 4, we show more of how this was achieved in code.

Corrupting a party is quite self-explanatory but an auxiliary effect is that the party's interpreter reveals its state to the adversary.

In **peeking into a buffer**, an adversary can learn the n-th message in the queue that is a buffer between a party and a functionality, however the message remains in the queue.

Clocking a message to a party $\mathcal{P}_i$ means the message in front of the relevant queue gets handed to the party's corruption module $\mathcal{Z}_i$. If the party is corrupted, it sends the message to the adversary, if honest, to the interpreter $\mathcal{I}_i$. However, **clocking a message to a functionality** $\mathcal{F}_k$ always results in the message being forwarded to the functionality for processing (since a functionality cannot be corrupted, as said above).

Querying the functionalities and **environment** were specified up to the input and output types of the calls to the functionality and environment.

This range of actions is arrived at in the original paper by showing that the behavior of an arbitrary adversary can be simplified this way. The adversary that can do only these actions (and fulfils some other criteria) is called a **lazy, semi-simplistic adversary**.

The other criteria are quite extensive in full. Among them is that a lazy adversary never desynchronizes a protocol by forcing a corrupted party to send a message before the protocol admits it. One of the semi-simplistic rules is that an adversary always forwards the message it gets from clocking an incoming buffer to the interpreter. There are checks at various points of the execution loop to make sure the adversary is lazy and semi-simplistic, and the result we later aimed to prove had to do with the forwarding of the clocked message.

4 Result and implementation

Since the result we aimed to prove informed, in big part, the model we built, the sections for implementation and result are the same. The workflow of the project was such that Sven would write sample code in Python that describes the protocol execution, and I would use that and the paper to build the Isabelle code.

4.1 Result

The result to prove was that in any possible state of the system, if an adversary clocks an incoming message to a party it has corrupted then it forwards that message to the interpreter without modifications. For that, it was necessary to build a loop as described below.

4.2 The code layout

The bulk of this work was to build the execution loop and what it requires. The behavior of the system $\mathsf{Env}\langle \mathsf{A}, \Pi \rangle$ is described so that a master scheduler – the adversary – is appointed. It enacts something on the system, the system reacts and gives a response back to the adversary. The adversary then decides on the next action by considering the system state and the input it received. Thus, the behavior of a protocol with an adversary can be viewed as a loop from one action of the adversary to the next; as an **execution**

loop

$$g : S \times A \to S \times A.$$

where S is the type of system states and A is the type of adversarial actions.

We describe the execution of a protocol in two phases – one phase in which the adversary performs an action on the protocol and the protocol responds with an input to the adversary; denote it as

$$f : S \times A \to S \times I;$$

and another phase where the adversary receives the input and responds to it with the next action. Then the two phases composed give the function g.

The task, then, was to model what the system does in response to the adversary, i.e. the function f. This entailed specifying the adversarial actions, the data they contain, and the response the action has on the system.

4.3 Datatypes

First, a range of datatypes had to be specified in order to create a model. In the system state, the parties and functionalities are described by two dictionary-like data structures that map the index of the machine to the machine itself. Thus, the code uses the types party_id and functionality_id a lot. They are synonyms to the type nat of the natural numbers – essentially, the identifiers are simply natural numbers, and the renaming simply clarifies the code.

Perhaps one more important datatype to mention is the write_instructions. The datatype write_instructions models the output of a party's interpreter, or a functionality or the environment after it processes a message. It shows which messages should be written to which outgoing buffers in response to the message.

4.4 Adversarial actions

An adversary can corrupt a party, clock and peek into both incoming and outgoing buffers, send messages on behalf of the corrupted parties, invoke the environment and query any functionalities.

Corrupting the party is uniquely defined by just one argument, which is the party's identifier. However, the other actions require more data to be uniquely defined. Thus, we introduce some custom datatypes.

The **buffer_action** type, for example, determines the party and functionality involved, the message index, as well as which type of buffer is meant and whether the buffer is clocked or just peeked.

```
datatype dir = Incoming | Outgoing
datatype act = Clock | Peek

record buffer_action =
  bufferParty :: party_id
```

```
  bufferFunc :: functionality_id
  bufferDirection :: dir
  bufferAction :: act
  bufferInd :: msg_index
```

The **send_message** type contains the party and functionality involved and the message sent:

```
record send_message =
  sendParty :: party_id
  sendFunc :: functionality_id
  sendDir :: dir
  sendMessage :: msg
```

Analogously, the datatype for **querying a functionality** is defined with a record containing the functionality_id and msg fields. All in all, the adversarial actions are defined with the datatype adv_action:

```
datatype adv_action =
  Empty |
  CorruptParty party_id |
  BufferAction buffer_action |
  SendMessage send_message |
  QueryFunctionality query_functionality
```

Once the adversary has chosen some action to take, the system reacts and gives a response to the adversary. The type adv_input describes the types of input the adversary can get from it:

```
datatype adv_input =
  AdvNone |
  CorruptionReply
    "(instance_label, instance_state * nat) map
    * public_param * private_param" |
  PeekReply msg |
  ClockIncomingReply "(nat * msg) option" |
  SendIncomingReply write_instructions |
  QueryFunctionalityReply msg
```

The first possible input to the adversary is AdvNone, the type for when the system gives no response to the adversary. This happens in the case of an invalid action (e.g. wanting to corrupt a party that is not listed), or in the case of clocking an outgoing buffer. The reply from corruption is the interpreter's internal state. Peeking a message gives the respective message, whereas clocking an incoming buffer might give the message that was clocked, but only if the party is corrupted. When an adversary sends an incoming message (on to the party's interpreter), it receives the interpreter's write instructions, and querying a functionaliy (here, also the environment), yields a message from the functionality.

4.5 Machines

A **protocol party** has buffers, a boolean field that shows whether the party is corrupted or not, and an interpreter.

```
record protocol_party =
    party_interpreter :: stateful_interpreter
    party_corrupted :: bool
    party_incoming_buffers :: "(functionality_id, msg list) map"
    party_outgoing_buffers :: "(functionality_id, msg list) map"
```

While in the theoretical model, the corruption module and interpreter are different machines, it was more natural to model the interpreter as contained in the protocol party in this case. The party also has functions related to corruption, communicating with the interpreter, and various buffer-related actions.

A party's **interpreter** contains its public and private parameters, and the state described above.

```
record stateful_interpreter =
  int_public_params :: public_param
  int_private_params :: private_param
  int_program :: "cmd list"
  int_incoming_buffers :: "int_msg list"
  int_outgoing_buffers :: "int_msg list"
  int_count_and_state :: "(instance_label, nat * nat * instance_state) map"
  int_port_count :: nat
```

It has internal semantics – rules for what to do with incoming messages depending on the rest of its state. All the semantics are defined as functions taking an interpreter and returning a new, changed interpreter.

The **functionality** is left rather unspecified. It is a record as follows:

```
record functionality =
  fnl_outgoing_buffers :: "msg list list"
  fnl_is_env :: bool
  fnl_state :: func_state
```

But in the loop, a functionality is not often altered, and so it does not have a lot of functions attached to it. Perhaps most importantly, it has a function that returns a message for the adversary when the latter queries it.

4.6 Execution loop

In the end, the execution loop $f : S \times A \to S \times I$ was built modularly: it was a function that contained a helper function for each adversarial action. Those helpers then

contained either the description of the system's response to that action, or other helpers that comprised the system's response.

Corruption and peeking messages were easy tasks to achieve. However, the biggest challenges were the clocking of messages, and the sending of messages. It would be too extensive to bring them out here in full, but the clocking and sending included multiple different actions (checks on the adversary's behavior, altering the buffers of a party, querying an interpreter, calling a functionality, and so on), in a way that was not always modular or in a way where return types could be understood in many ways. This made it an interesting challenge to translate from object-oriented code to functional code.

5 Conclusion and takeaways

While the project's scope was adjusted, the project was still quite work-intensive and intellectually challenging. It was hugely informative in terms of functional programming, but having chosen a smaller and more well-defined goal would have allowed to also reach the formal methods more.

Modelling something in functional code proved to require quite a bit of refactoring, and a lot of inquiry into the theory itself. Oftentimes, it was hard to refactor until the purpose of the system, both in what it does in real life, and what we aim to prove about it, was thoroughly clarified. It seemed sometimes that the ease of formal verification is immensely proportional to how well the system and the result are understood through a functional paradigm.

All in all, formal verification seems easier if the understanding of the problem, and the understanding of how formal verification works, are simple, well-defined and unambiguous. If the intentions aren't fully understood by the implementer, or the verification goal changes, a great deal of the model can become useless or need extensive refactoring. However, when built correctly, a formal model can be a very valuable tool to verify the properties of a protocol.

References

[BPW04] Michael Backes, Birgit Pfitzmann, and Michael Waidner. Secure Asynchronous Reactive Systems. 2004.

[CGB17] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. 2017.

[Fle16] Christopher Fletcher. *Oblivious RAM : from theory to practice*. PhD thesis, 2016.

[FM] Formal methods reading list. http://plfmse.cs.illinois.edu/formalmethods.html. Accessed: 2022-07-03.

[FNP04] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *EUROCRYPT*, 2004.

[NPW10] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. 2010.

[San12] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2012.

[Yao82] Andrew C. Yao. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science*, pages 160–164, 1982.

A Output equivalence with shared memory model

A.1 Brief description of the result

The result whose proof we previously aimed to verify is about a protocol Π being as secure as an altered protocol $\Pi^{\diamond}$. This is shown by taking a certain adversary A, giving a construction $\phi_{\diamond}(\mathsf{A})$ of a novel adversary and showing that for any environment Env, the collections $\mathsf{Env}\langle\Pi, \mathsf{A}\rangle$ and $\mathsf{Env}\langle\Pi^{\diamond}, \phi_{\diamond}(\mathsf{A})\rangle$ are output equivalent. After introducing the security definitio, bisimulations, and the other model, the initial result is stated. To finish off, we sketch how the bisimulation argument would have been adapted to the model(s) we would have built.

A.2 The security definition of a protocol

The information contained in this subsection is a recap of the material in the aforementioned, not available yet doctoral thesis of Pille Pullonen-Raudvere.

Let there be three collections Env, A and Π that model, respectively, the protocol Π, the environment Env calling out the protocol Π, and the adversary A. If the interface of Π splits into two subinterfaces – one to match with the interface of Env and one with

A – then we can view a collection $\mathsf{Env}\langle\Pi, \mathsf{A}\rangle$. The security definition is given for such closed collections.

Multiple notions of equivalence can be given for two such closed collections. The one used in the result to be proved is of output equivalence; this means that the outputs given by Env are indistinguishable in the closed collections $\mathsf{Env}\langle\Pi_1, \mathsf{A}_1\rangle$ and $\mathsf{Env}\langle\Pi_2, \mathsf{A}_2\rangle$.

The formalisation of the security definition is as follows. Let Π_1 and Π_2 be collections with an identical service interface and let $\mathbb{E}$ be the set of compatible environments. Let $\mathbb{A}_1, \mathbb{A}_2$ be the set of possible adversaries for Π_1 and Π_2, respectively. Then Π_1 is as secure as Π_2 if there is a construction $\rho : \mathbb{A}_1 \to \mathbb{A}_2$ such that

$$\forall \mathsf{A}_1 \in \mathbb{A}_1, \forall \mathsf{Env} \in \mathbb{E} : \mathsf{Env}\langle\Pi_1, \mathsf{A}_1\rangle \equiv \mathsf{Env}\langle\Pi_2, \rho(\mathsf{A}_1)\rangle.$$

Essentially, what this says is that Π_1 is as secure as Π_2 if, for any adversary A_1 and environment Env that one can compatibly connect to Π_1, the collections $\mathsf{Env}\langle\Pi_1, \mathsf{A}_1\rangle$ (the collection of joining Env and the adversary A_1 to Π_1) and $\mathsf{Env}\langle\Pi_2, \rho(\mathsf{A}_1)\rangle$ (the collection of joining Env and the altered adversary $\rho(\mathsf{A}_1)$) are equivalent.

A.3 Bisimulation proofs

Let us define a labelled transition system and a bisimulation – we will use related notions in constructing the proof of the result in Isabelle. [San12] A **labelled transition system** is a triple (S, Λ, σ) where S is called the set of states, Λ is called the set of actions, and $\sigma \subset S \times \Lambda \times S$ is called the transition relation. If there is a triple (s, α, s') in the transition relation, we denote it also as $s \xrightarrow{\alpha} s'$. The definition of a bisimulation says that if two states s and t of a system are in relation, and there exists a transition α from s to s', then the same transition also exists out of t, and moreover, the yielded new states s' and t' are also related; analogously for if a transition exists out of t. This creates a sort of "lockstep" changing of the states where the transitions out from a pair of related states match, and one cannot reach an unrelated pair of states from a related pair. [San12] Let $\mathcal{L} = (S, \Lambda, \sigma)$ be a labelled transition system, and let $R \subset S \times S$ be a relation on the states of $\mathcal{L}$. The relation R is a **bisimulation** if whenever $(s, t) \in R$,

$$(\exists \alpha \in \Lambda, s' \in S : s \xrightarrow{\alpha} s') \implies (\exists t' \in S : t \xrightarrow{\alpha} t') \wedge (s', t') \in R$$

and

$$(\exists \alpha \in \Lambda, t' \in S : t \xrightarrow{\alpha} t') \implies (\exists s' \in S : s \xrightarrow{\alpha} s') \wedge (s', t') \in R$$

The union of two labelled transition systems is itself a labelled transition system. Thus, while we consider the relation R on the states of a single LTS, it is still possible to consider it on two LTS-s by taking their union and considering the respective relation.

A.4 Shared-memory model

The first model is the one described above. In the second, "shared-memory" model, the protocol party $\mathcal{P}_i$ is now split into three: the **interpreter** $\mathcal{I}_i^\diamond$, the **corruption module** $\mathcal{Z}_i^\diamond$, and the **memory** $\mathcal{M}_i^\diamond$. Now $\mathcal{I}_i^\diamond$ is a stateless interpreter and $\mathcal{M}_i^\diamond$, the memory, is

a dedicated machine where the internal state of the interpreter including all random choices is stored. The **functionalities** now also have access to the shared memory modules $\mathcal{M}_i$, and a new functionality $\mathcal{F}_{io}^{\diamond}$ is introduced.

The protocol $\Pi^{\diamond}$ is considered to be the interpreters, shared memories and functionalities, whereas the adversarial construction $\phi_{\diamond}(\mathsf{A})$ will now also include the new corruption modules $\mathcal{Z}_i^{\diamond}$. The corruption modules' purpose is to simulate the behavior of the interpreter to the adversary, and the proof of the theorem relies in big part on constructing such an adversary that also abides by certain additional constraints on the protocol.

A.5 Output equivalence of the basic and shared-memory models

The theorem that we wish to verify is as follows. Let Π be the protocol with a well-formed implementation and let E_{Π} be the set of compatible environments. Then

$$\forall \mathsf{Env} \in \mathbb{E}_{\Pi} : \forall \mathsf{A} \in \mathbb{A}_{\Pi,\mathsf{Env}}^{lazy} : \quad \mathsf{Env}\langle \Pi, \mathsf{A} \rangle \equiv \mathsf{Env}\langle \Pi^{\diamond}, \phi_{\diamond}(\mathsf{A}) \rangle,$$

where $\mathbb{A}_{\Pi,\mathsf{Env}}^{lazy}$ is the set of compatible lazy, semi-simplistic adversaries. In other words, this is a specific case of the general security definition A.2; we wish to show that the shared-memory protocol $\Pi^{\diamond}$ corresponding to Π is as secure as Π.

Note that the protocol is considered well-formed if it has certain properties regarding the correct use of memory, the correct order of messages, and restrictions on how memory locations can be accessed or written into.

A.6 Proving modified bisimulation

In the approach to verifying the proof, a property similar to being a bisimulation is used. Let us introduce a definition for it. Let $\mathcal{L} = (S, \Lambda, \sigma)$ be a labelled transition system, and let $R \subset S \times S$ be a relation on the states of $\mathcal{L}$. Take $(s, t) \in R$ and *an adversarial action* a. Call R a **modified bisimulation** if

$$((s', o) \leftarrow f(s, a) \wedge (t', p) \leftarrow f(t, a)) \implies (s', t') \in R \wedge o = p.$$

This definition of a bisimulation more aptly helps model output equivalence of our protocols. It says that if there is an adversarial action that takes a related pair of states (s, t) to the states s' and t', then those states are also related, i.e. $(s', t') \in R$, and the inputs provided to the adversary o and p are the same.

Then, for a proof we need to show that the pen-and-paper relation fulfills two conditions: that the initial states of both systems are related, and that it is indeed such a modified bisimulation. Doing those two things ensures that for any action the adversary might take on either system, the systems execute in a precisely analogous manner, providing the exact same inputs to the adversary. Thus, the protocols are output equivalent.