Project Report

Gianni Perlini, Andrea Rinaldi

Lecturers: Dr. Ralf Sasse, Dr. Christoph Sprenger

May 29, 2018

Abstract

In this project we examine the behaviour of two key exchange protocols used in practice to secure machine readable travel documents and secure messaging. To do that, we formalize the protocols using the TAMARIN prover and use the tool to verify or falsify the security properties that the protocols should achieve. The process is done by refinement steps. We organize this report in four sections, namely *Introduction*, *PACE Protocol*, *OTR Protocol* and *Conclusions*.

1 Introduction

In this section we give a brief introduction to the protocols we are going to formalize.

1.1 Password Authenticated Connection Establishment Protocol

This protocol is part of the protocol suite used for machine-readable travel documents, such as, for example, e-passports and identity cards. PACE enables the RFID chip and the terminal to establish a secure communication channel starting from a low-entropy shared secret, which is used in combination to the Diffie-Hellman key exchange protocol to derive a strong shared session key. The initial secret can either be a **P**ersonal **I**dentification **N**umber (**PIN**) only known by the owner of the document, the **M**achine **R**eadable **Z**one (**MRZ**) printed on the document, or a six-digits **C**ard **A**ccess **N**umber (**CAN**) [1], [3].

1.2 Off-The-Record Protocol

OTR is a protocol designed to provide security features for instant messaging applications. This protocol consists of two phases. The first one is an authenticated key exchange which allows the parties to generate a shared session key. The second one consists of a continuous refreshment of the session key during the instant messaging exchange [1], [2], [4]. In our project we only considered the first phase.

2 PACE Protocol

In this section we discuss the formalization of the PACE protocol as well as our findings regarding the security properties we tried to verify.
This section is further divided into five subsections, corresponding to the five refinement steps that we used to formalize PACE.

2.1 A simple challenge-response protocol

The first protocol we are going to formalize is a simple challenge-response protocol between an initiator A and a responder B which we can illustrate as follow in *Alice&Bob* notation:

$$A \to B : x$$
$$B \to A : [x]_{k(A,B)}$$

In this protocol, A generates a nonce x and sends it to B, which then responds sending a **M**essage **A**uthentication **C**ode (**MAC**) of the nonce. We write this MAC as $[x]_{k(A,B)}$ where $k(A, B)$ is an uni-directional long-term symmetric key shared between the two participants. We want to verify that this simple protocol achieves injective agreement of A with B on nonce x.
The formalization of this first step is quite straightforward and can be found in `PACE/P1.spthy`. We begin by setting up the long-term key (`Shared_Key` rule) and by specifying that the attacker can act as a protocol participant (`Reveal_SK` rule), as asked in [1]. We also make sure that the key is indeed uni-directional by adding a TAMARIN restriction, called `Inequality`, to the theory file, and by enforcing it in the shared key set-up. We then proceed by initializing the two parties and by formalizing, for each of the two messages exchanged, a send rule and a receive rule, for both A and B. In order to let the parties be able to distinguish the exchanged messages more easily, and as advised in [1], we add tags to each one of the messages. Together with lemmas to verify agreements, we also include an executability one, called `executable`, which ensures that the model is actually executable. This lemma will be present in all our files.
Figure 1 shows the formalization of the first step, including the `Running` and `Commit` claims needed to verify agreement.
Running TAMARIN on `P1.spthy` indeed prove that our first step protocol achieves injective agreement as wanted, as can be also seen in `P1-proof.spthy`. This is what we would expect since the initiator receives a MAC of its nonce computed with the secret key which is only known by B.

2.2 Mutual authentication

As a first refinement, we combine two instances of the first step protocol to produce the following one:

$$A \to B : x$$
$$B \to A : y$$
$$A \to B : [y]_{k(A,B)}$$
$$B \to A : [x]_{k(B,A)}$$

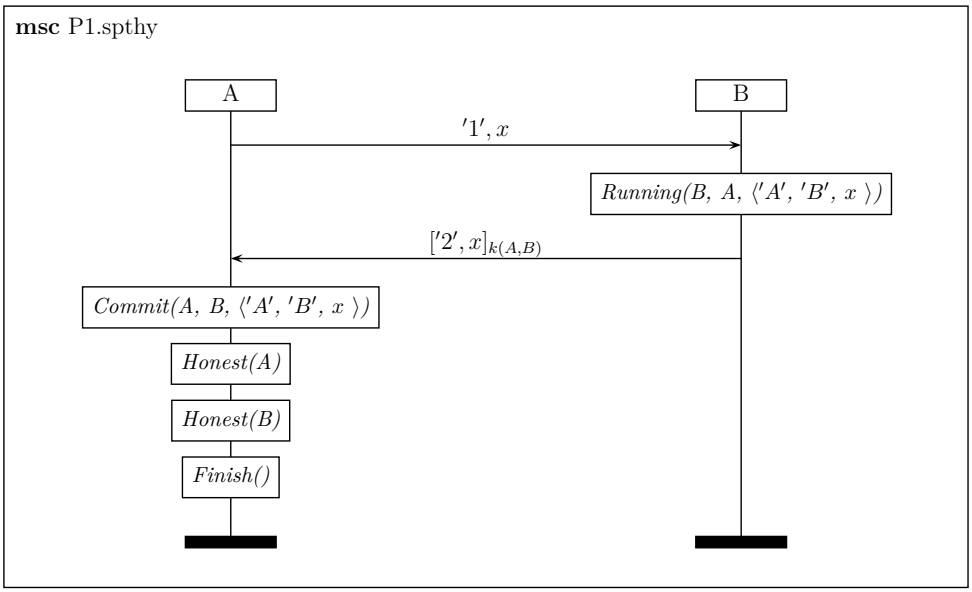

Figure 1: P1 specification

Note the different keys used for the different directions, since as specified in the previous subsection, keys are uni-directionals.
Recall that the previous protocol only achieved injective agreement on x from the initiator's point of view. In this refinement step, we would like to achieve agreement for both points of view on both the nonces. To do that, we specifiy the protocol as follows:

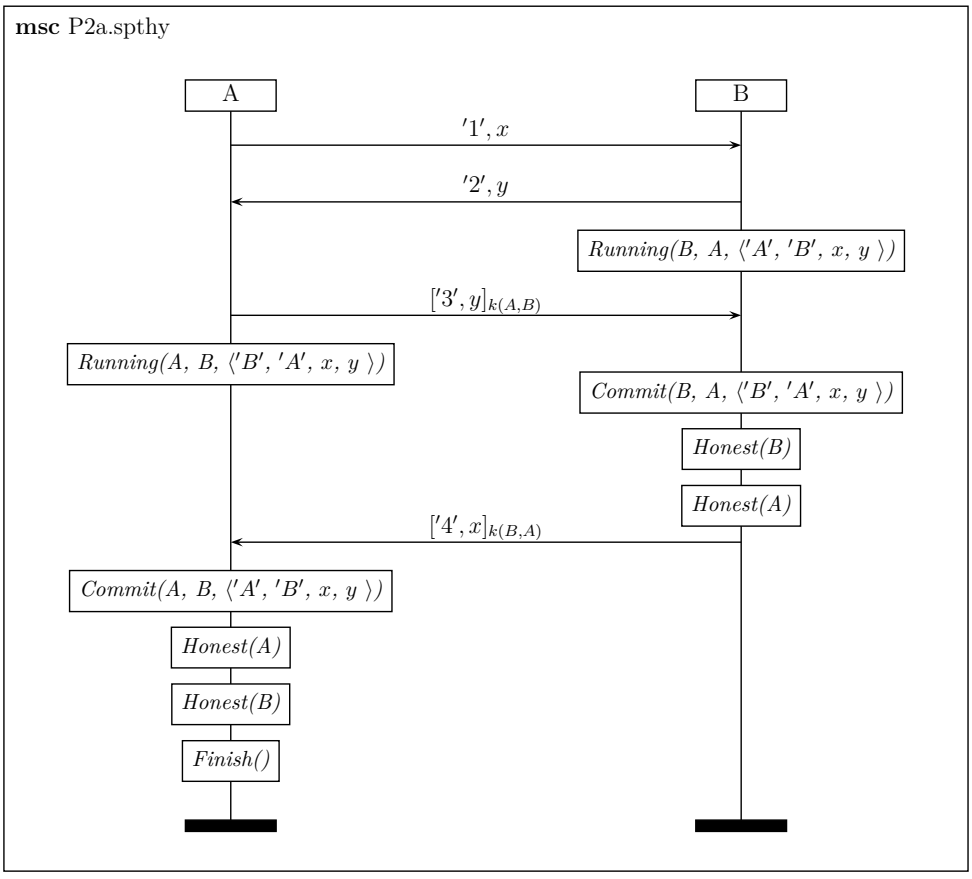

Figure 2: P2a specification

Running the prover on the model of *Figure 2*, we can see that not only this protocol does not satisfy non-injective agreement from the responder's point of view (and thus it does not satisfy injective agreement neither), but we also lose the property on the initiator's side. The attacks found by TAMARIN can be seen in the `P2a-proof.spthy`. We also included images of the attacks in the `PACE` folder. The first attack proves that non-injective agreement does not hold on A's side, and can be represented as follows in *Alice&Bob* notation:

$A \rightarrow I(B) : x$
$I(B) \rightarrow A : y$
$I(A) \rightarrow B : x$
$B \rightarrow I(A) : y'$
$I(B) \rightarrow A' : y'$
$A' \rightarrow I(B) : [y']_{k(A',B)}$
$I(A') \rightarrow B : [y']_{k(A',B)}$
$B \rightarrow I(A') : [x]_{k(B,A)}$
$I(B) \rightarrow A : [x]_{k(B,A)}$
A *commits on* $\langle x, y \rangle$ *without corresponding* `Running` *claim*

Note that in order to perform step 5, A' acts as an initiator by sending a first message which is discarded by the attacker.
The second attack proves that the same properties do not hold from B's point of view neither. In this case, the attacker acts as the initiator and sends a random nonce x to B, which responds with y. The attacker takes advantage of another initiator A which sends its nonce x' discarded by the attacker. The latter then forwards y to A, which responds with a valid MAC, which is simply forwarded to B to conclude the attack.

$I(A) \rightarrow B : x$
$B \rightarrow I(A) : y$
$I(B) \rightarrow A : y$
$A \rightarrow I(B) : [y]_{k(A,B)}$
$I(A) \rightarrow B : [y]_{k(A,B)}$
A *running on* x' *and* y
B *commits on* x *and* y

To propose a solution to overcome these attacks and achieve the wanted security properties, it is important to understand the problem underlying the attacks. In both cases, the adversary takes advantage of the fact that, even if it does not know the key, it can trick the other party into computing a valid MAC on A's (respectively B's) nonce. Given this insight, it is easy to see that the attack would not work if the MAC would be computed on both the nonces, since in the first case A would notice that $y \neq y'$ and in the second case B would notice $x \neq x'$. A naive implementation of this solution would be to mac all messages. However, this would be a waste of computation since it is sufficient to modify messages 2 and 3 in order to respectively include x and y on the MAC. *Figure 3* shows the fixed protocol as we specified it in TAMARIN. Note that we only include messages 3 and 4 since the previous ones remain unchanged, and we also omit the claims since they are placed at the same positions as in *Figure 2*.
As can be seen in file `P2b-proof.spthy`, this version of the protocol indeed achieves injective agreement (and thus non-injective agreement too) for both nonces from both parties' points of view.

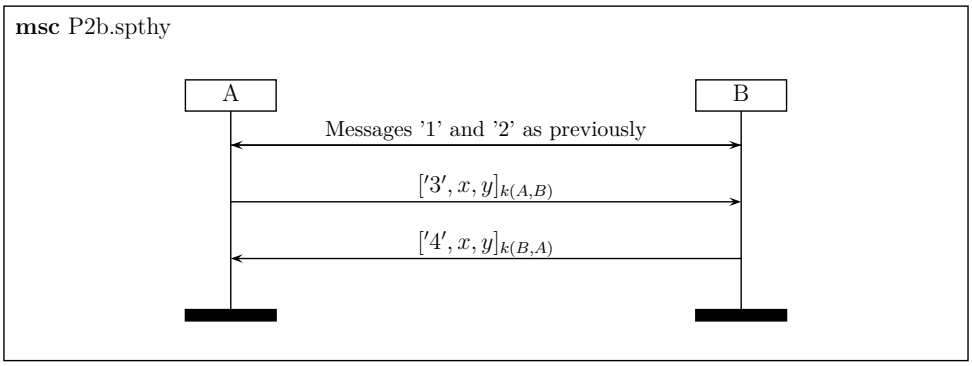

Figure 3: Fixed protocol. This is theory P2b

2.3 Introducing a session key

The next refinement step consists of introducing a session key generated from the long-term key and the two nonces, as well as a related secrecy property. The session key is used to compute the MAC and is generated as follows [1]:

$$Kab = kdf(k(A, B), x, y)$$

where `kdf` denotes a **K**ey **D**erivation **F**unction which is implemented in TAMARIN using a user-defined function. The protocol we want to specify is thus the following:

$A \to B : x$
$B \to A : y$
Generate $Kab = kdf(k(A, B), x, y)$
$A \to B : [y]_{Kab}$
$B \to A : [x]_{Kab}$

Note that this protocol introduces the session key on the one specified by `P2a`, where each participant only macs the other party's nonce. Again, we want to verify that this protocol achieves injective agreement on nonces and the session key, and additionally secrecy on Kab. We specify the protocol as depicted in *Figure 4*.
Running the prover on `P3a.spthy` we indeed verify that this protocol satisfies all the security properties we wanted it to provide. An important and worth investigating difference between this protocol and the previous one is that the parties agree on both nonces, in spite of one of them not being maced by the owner. In fact, if we look at *Figure 2* and *Figure 4*, we can notice that the protocol is almost the same, except that now the MACs are computed using Kab instead of the long-term key. This difference is actually the one that makes it possible for protocol `P3a` to provide injective agreement. Since the session key is derived from the long-term key and both the nonces, this implies that the correct nonces have been used as inputs of the `kdf`, otherwise the MAC would not be accepted. In this case, the attacker can't do anything to threaten injective agreement without compromising the long-term key (and thus, the session key).

2.4 Replace the password by a nonce

We now modify the derivation of the session key to include a fresh nonce generated by the initiator, which replaces the long-term key as follows:

$$Kab = kdf(s, x, y)$$

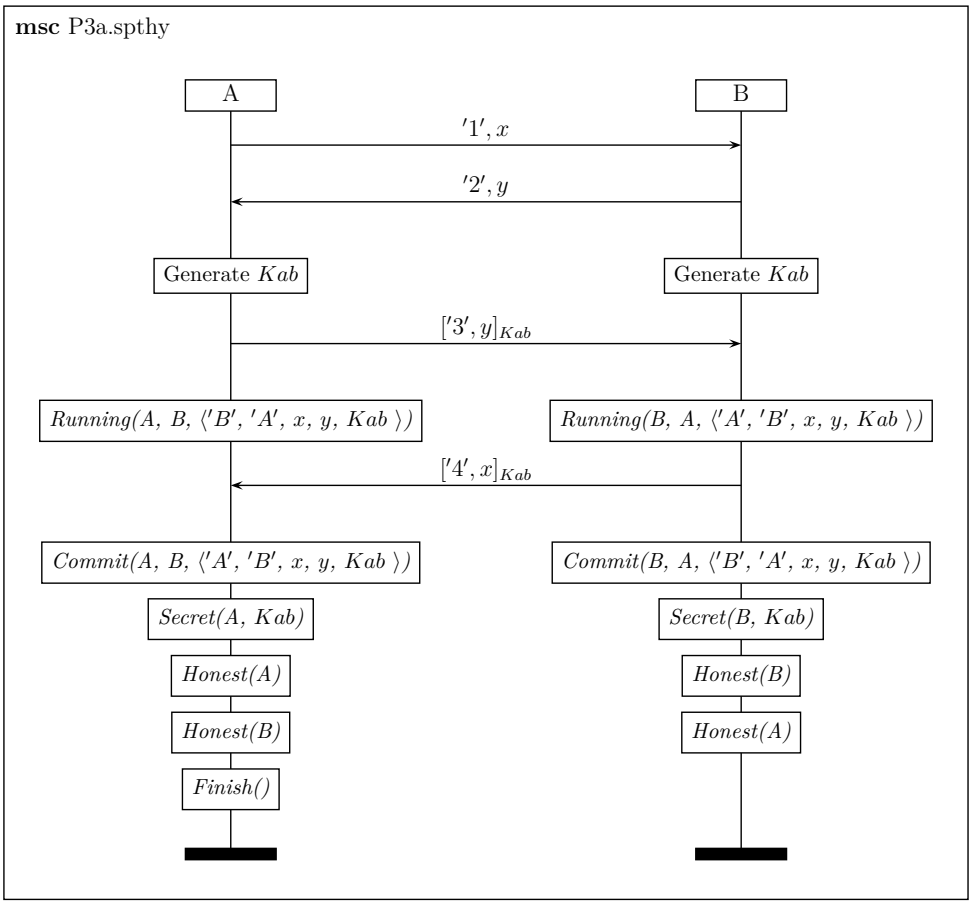

Figure 4: P3 specification

Where s is the new nonce generated by A and sent to B encrypted using the hash of the long-term key. Hence, the first message becomes

$$A \to B : x, \{s\}_{h(k(A,B))}$$

Figure 5 represents the protocol `P4.spthy`. We use a TAMARIN built-in function to deal with hashing.
As can be seen from the `P4.spthy` file, this modification does not compromise the security properties of the protocol, which still get verified by TAMARIN.

2.5 Introducing Diffie-Hellman: The PACE protocol

This is the final refinement step, which gives us the complete PACE protocol. The refinement consists of replacing both nonces x and y with half Diffie-Hellman keys, namely g^x and g^y, where the generator g is chosen using the nonce introduced in the previous step, *i.e.* s, and a public parameter p, both given as inputs to the mapping function `map`. After exchanging the two half-keys, the parties generate the session key as

$$Kab = h(g^{xy}) = h(g^{yx})$$

We use a user-defined function to declare the mapping. Also, the public parameter p is included in plain text in the first message. This last step is defined in `P5ab.spthy` and is depicted in *Figure 6*.

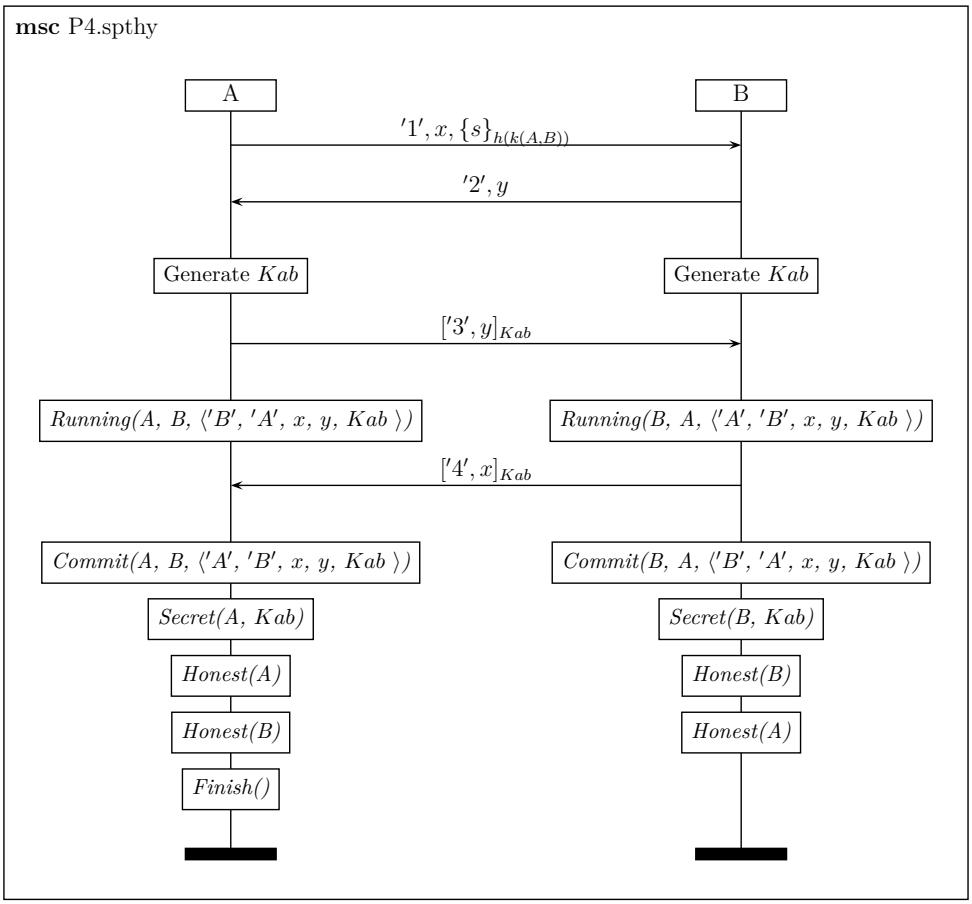

Figure 5: P4 specification

This final protocol satisfies all the properties as the previous one did, as well as an additional one, namely **P**erfect **F**orward **S**ecrecy (**PFS**). Note that in *Figure 6*, in the *Running* and *Commit* claims, A (respectively B) cannot be sure that the received parameter is indeed g^y (g^x respectively). This is why claims include the value gX and gY instead of the half-keys. It is important to note that this protocol relies on the secrecy of the base g in order to provide the specified properties. If the base was public, the protocol would suffer from the usual *Man-in-the-Middle* attack that can be mounted on anonymous Diffie-Hellman. To see this, consider the protocol with g being public. The s and p parameters would not be used any more since they were given as inputs to the mapping function returning the base, which is now public. This means that an attacker could simply interact with both A and B and generate two separate session keys g^{xi} and g^{iy}. This is possible because the two half-keys are not authenticated. To solve this problem while still using a public base, the protocol should use authenticated Diffie-Hellman, for example by means of certificates as done in `TLS`.

To conclude this first part of the project, we show an attack on non-injective agreement found by TAMARIN when we do not include tagging in messages. In this case, the last two messages become unifiable, and can be exploited by an attacker to trick A into thinking to be talking with B, while talking to him/herself. The attacker performs a reflection attack by taking the first message and sending it back to A, discarding the s and p parameters. Since A does not check if the received value is the same as the one sent out to B, and since g^x is random as g^y would be, the attack is not detected and the protocol is continued. The same happens with message 3, which is reflected to A. The attack is

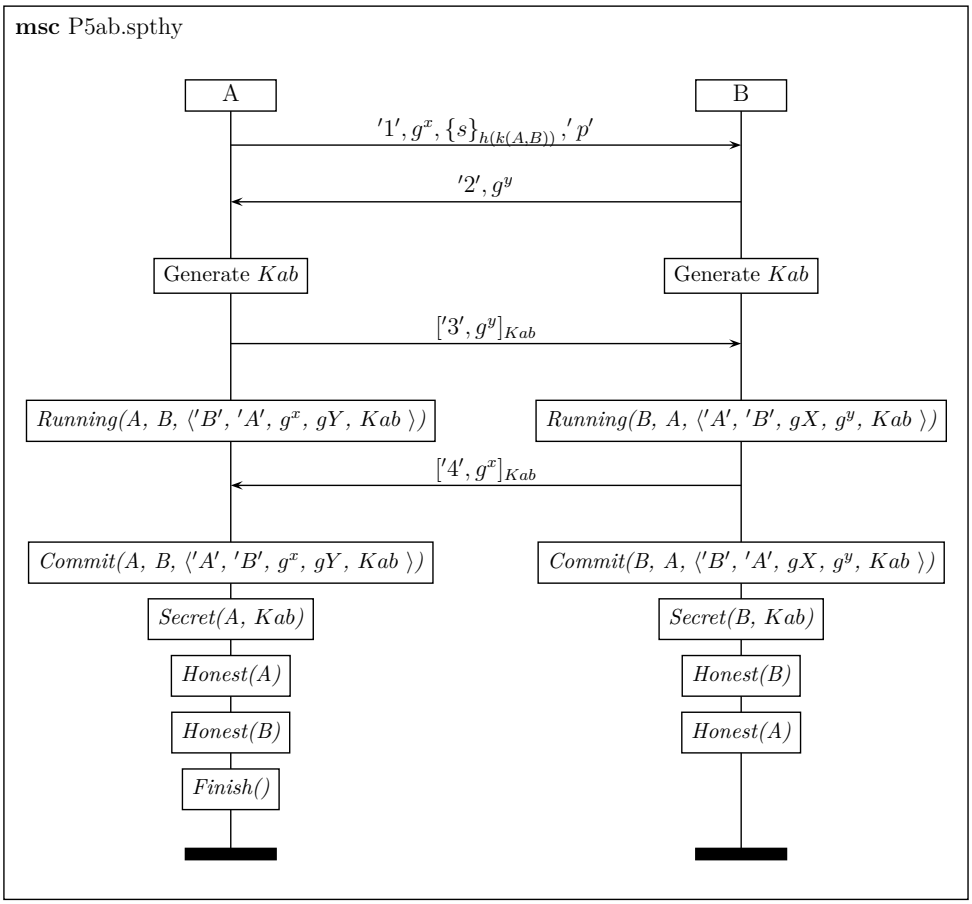

Figure 6: P5ab specification

represented in `P5d-NIA-attack.png`, which can be found in `/PACE/Proofs`. To solve the problem without introducing any tags, we can use the same restriction that we used to ensure that the long-term keys are unilateral, namely the `NotEq` one. Inserting it into the `A_2_Receive` rule as an action `NotEq(`g^x`, `gY`)` solve the issue and now all the properties get verified by the prover. The non-fixed version is specified in `P5dAttackable.spthy`, whereas the corrected one is `P5d.spthy`.

3 The Off-the-Record Messaging Protocol

In this section we discuss the formalization of the OTR protocol as well as weaknesses found using TAMARIN. We proceed in 4 steps. First, we formalize the original protocol and we demonstrate an authentication failure as described in *Section 2.1* and *Section 3.1* of [2]. We then improve the protocol as discussed in *Section 3.1* and we finally model the `SIGMA-R` protocol as described in *Section 4.1* of the same paper.

3.1 Modelling the original OTR key exchange

In this first step we model the original protocol. The authenticated key exchange phase takes place as follows:

$$A \to B : \{g^x\}_{skA}, pkA$$
$$B \to A : \{g^y\}_{skB}, pkB$$
$$\text{Generate } Kab = g^{xy}$$

where $\{x\}$ indicates a digital signature on x using the party's private key.
The corresponding protocol as specified in the prover is depicted in *Figure 7*.

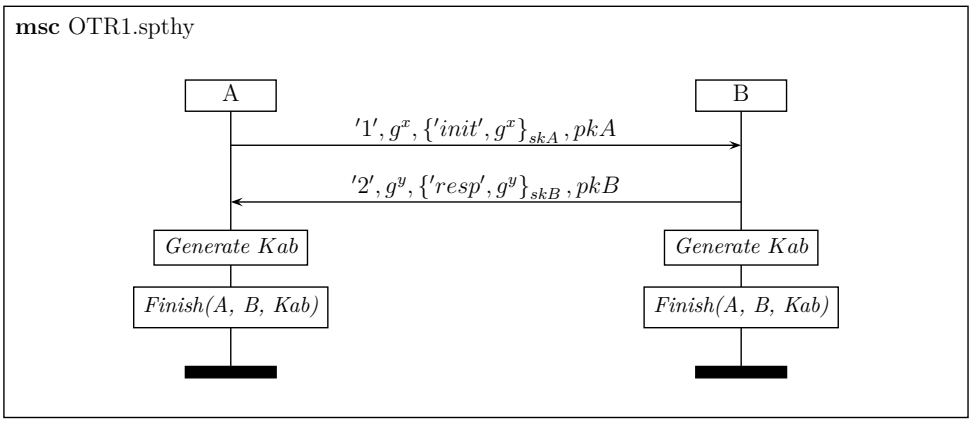

Figure 7: MSC of `OTR1.spthy`

Note that we omitted all claims since we do not discuss security properties for this simple protocol. Here we limit ourselves to discuss the differences between our formalization and the actual protocol and to informally discuss its correctness. Also note that the only claim we included is `Finish`, which is used as previously in the `executable` lemma. Note, however, that this is different from before. In this case, and in the the rest of this report, we want the trace to contain a `Finish` claim from both users and that they "agree" on a common term (namely, the session key). This, together with the restriction that the parties cannot be executed more than once (`Unique_Init` in `Init_A/B rules`), is done in order to verify that the protocol is executable without the help of the attacker.
First of all, as we have done previously, we add tagging to each one of the messages exchanged by the peers to avoid messages to be unifiable. We also include this tagging inside the signatures since the first part is in plaintext and can be modified by the attacker. Another difference to notice is the fact that the peers send their public value g^i in clear in addition to the signed version. This is done in order to allow the recipient of the message to verify the signature. In TAMARIN, we did it using the following statement (for B in this case):

```
Eq(verify(signature, <'init', gX>, pkI), true)
```

where `signature` and `gX` represents B's view of both A's signature and half-key, and `Eq` is a restriction which ensures that the two parameters are equal.
Besides the described differences, the specified protocol behaves the same as the original one. We first setup the PKI and allow the adversary to reveal the long-term keys of the parties (`Register_pk` and `Reveal` rules). We then initialize A and B using the `Init_*` rule where we create the agents with initial states containing the thread ids, the agents' name and their public/private key pairs. The rest of the protocol is formalized in a straightforward manner: the first message is sent by A and received by B using the `A_1_Send` and `B_1_Receive` rules, which use the `built-in` functions `signing` and `diffie-hellman` to deal with signatures and half-keys. The second message is specified by `B_2_Send` and `A_2_Receive` rules, which make sure that both parties generate and store the session key and destroy the relative ephemeral parameters.

3.2 Authentication failure

In this step we analyse the security properties of the protocol specified in the previous section. In particular, we want to write a lemma that allows TAMARIN to show a trace corresponding to the attack described in *Section 3.1* of [2]. To do that, we first describe the attack and then show the lemma we used to find the corresponding trace.
The attack, in *Alice&Bob* notation is the following:

$$\begin{aligned} A \to I(B) &: g^x, \{g^x\}_{skA}, pkA \\ I(A) \to B &: g^x, \{g^x\}_{skI}, pkI \\ B \to I(A) &: g^y, \{g^y\}_{skB}, pkB \\ I(B) \to A &: g^y, \{g^y\}_{skB}, pkB \end{aligned}$$

The attacker simply signs A's half-key with its private key, and sends the signature along with g^x and the corresponding public key. It then forwards B's response to A. At the end of the protocol, A and B agree on the private key g^{xy}, which the attacker cannot compute, but while A thinks to be talking to B, the latter is convinced to be talking to the intruder.
The following figure shows the protocol as specified in `OTR2.spthy`. The protocol is the same as the one shown in *Figure* 7, with the addition of the `Running` and `Commit` claims that are used in the lemma to find the attack.

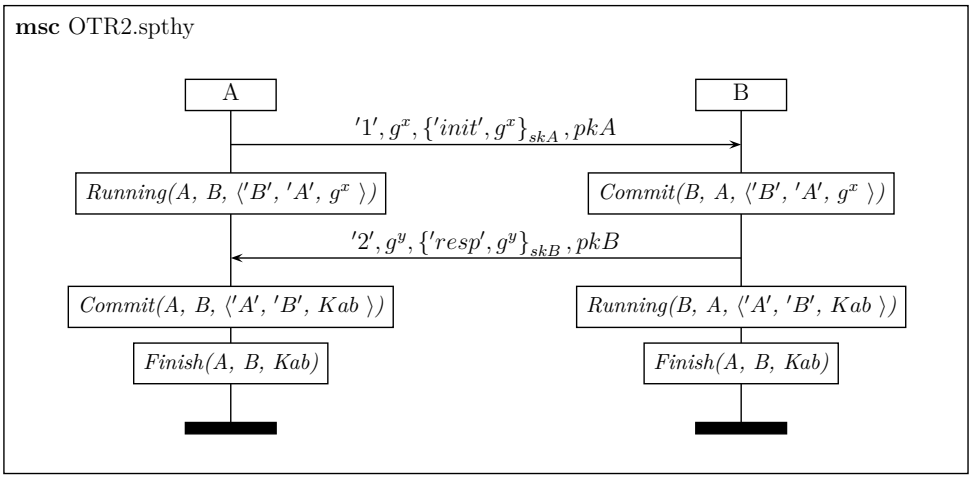

Figure 8: OTR2 specification

Note that we omitted the `Secret` and `Honest` claims as well as the key generation action. Also note that in the sequence chart, the `Running` and `Commit` claims seem to happen at the same time. However, the former is specified when the message is sent, whereas the latter happens when the message is received.
We now show the lemma we used to let TAMARIN find the attack described in the paper, and we discuss it.

```
lemma attack:
  exists-trace
    "
      Ex a b c t #i #j.
         Running(b, c, <'I', 'R', t>) @j
       & Commit(a, b, <'I', 'R', t>) @i
       & #j < #i
```

```
    & not (Ex #k. Reveal(a) @k)
    & not (Ex #u. Reveal(b) @u)
    & not (Ex id #v. Create_A(c, id) @v)
    & not (Ex idd #w. Create_B(c, idd) @w)
"
```

We want the tool to find a trace where the conditions are satisfied so we begin with the keyword `exists-trace`. We require the lemma to include a `Commit` claim by A with B on a term t (the key Kab in our case), which must be preceded by a `Running` claim made by B with another party C, which is the intruder, on the same term. This captures the fact that in the attack the initiator thinks to be talking to B while the latter thinks to be talking to the intruder. Moreover, we require the parties A and B not to be compromised. Finally, we require that the third party C has not been initialized (no `Create_A/B` actions in the trace). This is done in order to enforce that the third party C is indeed the attacker. As can be seen in `OTR2_paper_attack.png` in `OTR/Proofs`, the attack depicted in the figure is the same we described above.

3.3 Improvement

In this step we implement the improvement described in *Section 3.1* of [2]. This is again a two message protocol, we show it in *Alice&Bob* notation below.

$$A \to B : g^x, \{g^x, B\}_{skA}, pkA$$
$$B \to A : g^y, \{g^y, A\}_{skB}, pkB$$
$$\text{Generate } Kab = g^{xy}$$

As can be seen, this is a slightly modified version of the protocol we discussed in 3.1, and consists of inserting the identity of the intended receiver in the signature. However, as discussed by the author of [2], this modification completely breaks deniability since the identities of both peers are digitally signed and thus non-repudiable.
We next discuss the formalization of this protocol in TAMARIN and show that this is not the only problem. *Figure 9* depicts the protocol as we formalized it with the tool.
Again note that the `Running` and `Commit` claims are specified in the `Send` and `Receive` rules respectively.
As can be seen inspecting `OTR3-proof.spthy`, running the tool on this specification shows that the protocol only achieves secrecy (and provides PFS) and non-injective agreement from the responder's point of view. We now describe the attacks on non-injective agreement from A's point of view (the injective agreement's attack is of course the same) and on injective agreement on B's side. These two attacks are depicted in the corresponding figures that can be found in `OTR/Proofs`.

- *Non-Injective Agreement A*: The attacker exploits two different runs of the protocol by A, we call them A and A'. A sends the first message which is blocked by the attacker. The first message arriving at B is the one sent by A' with half-key $g^{x'}$. The responder then sends the second message containing g^y, which is redirected to A by the attacker. At the end, A computes g^{xy} while B computes $g^{x'y}$. Note that the identity in the messages are not helping since the peers A and A' are the same, but are running the protocol at different points in time. We show the attack using a msc diagram where we only show the half-keys exchanged for simplicity.

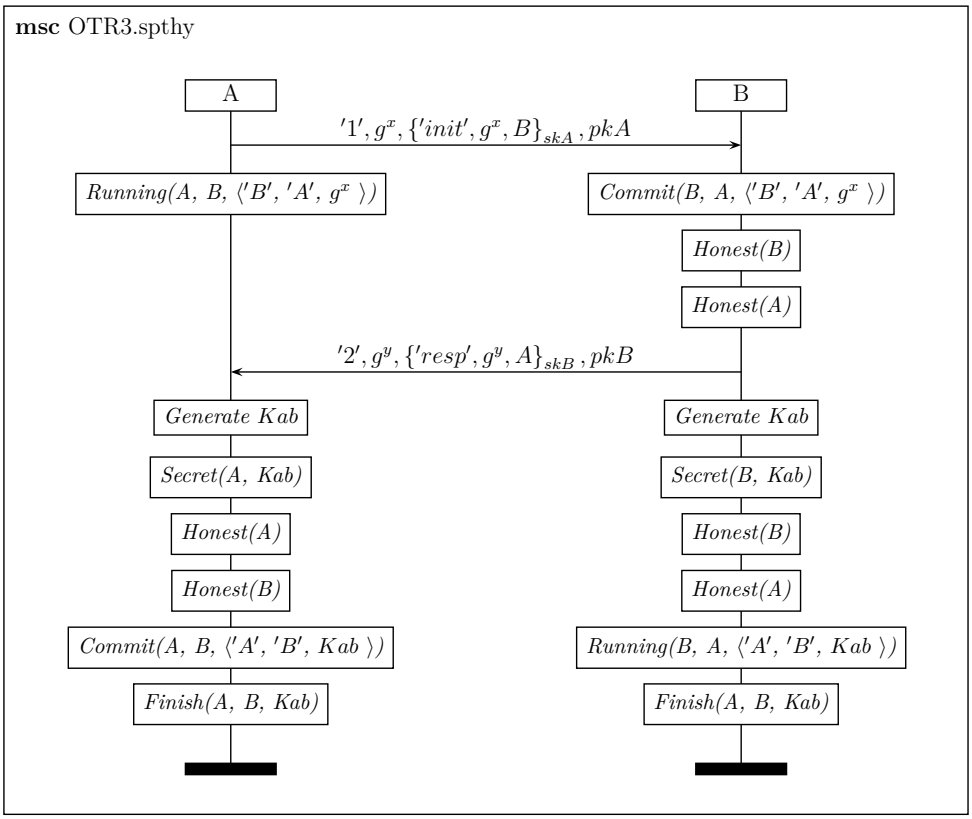

Figure 9: OTR3 specification

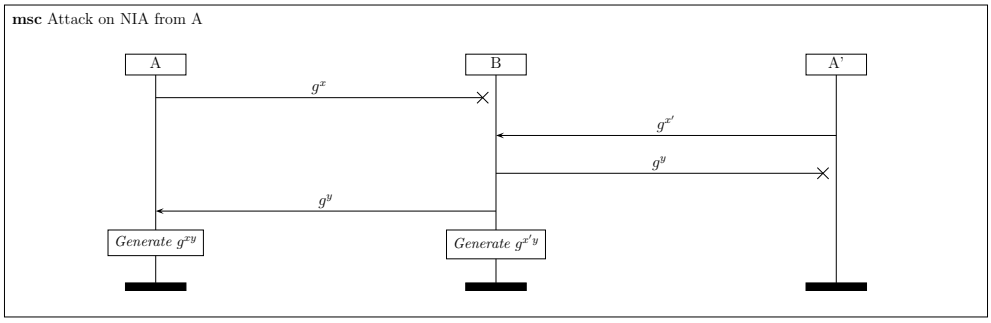

Figure 10: NIA attack from A's point of view.

- *Injective Agreement B*: In this case the attack is fairly simple. We have two instances of the responder B and B' and one of the initiator A. The latter sends the first message with g^x to B. The attacker intercepts the message and, since there is no freshness involved in the protocol, sends it in both runs of the responder, meaning that for a single `Running` claim from A, there are two identical `Commit` claims from B.

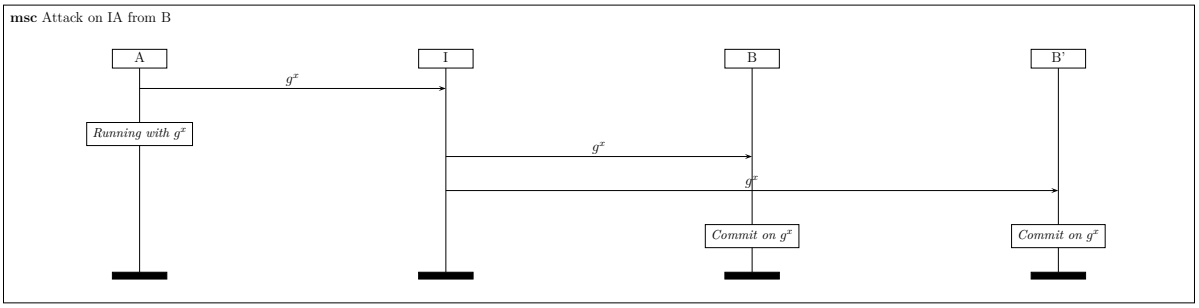

Figure 11: IA attack from B's point of view.

3.4 SIGMA

In this last step we formalize the `SIGMA-R` protocol described in [2]. The protocol behaves as follows:

$$
\begin{aligned}
&A \to B : g^x \\
&B \to A : g^y \\
&\text{Generate } K_m = h(g^{xy}) \\
&A \to B : A, \{g^y, g^x\}_{skA}, [\,'0', A]_{K_m}\ pkA \\
&B \to A : B, \{g^x, g^y\}_{skB}, [\,'1', B]_{K_m}\ pkB
\end{aligned}
$$

Where $[x]_k$ denotes the MAC of x with key k. *Figure12* represents our formalization of `SIGMA-R`, which can be found in `OTR4.spthy`.

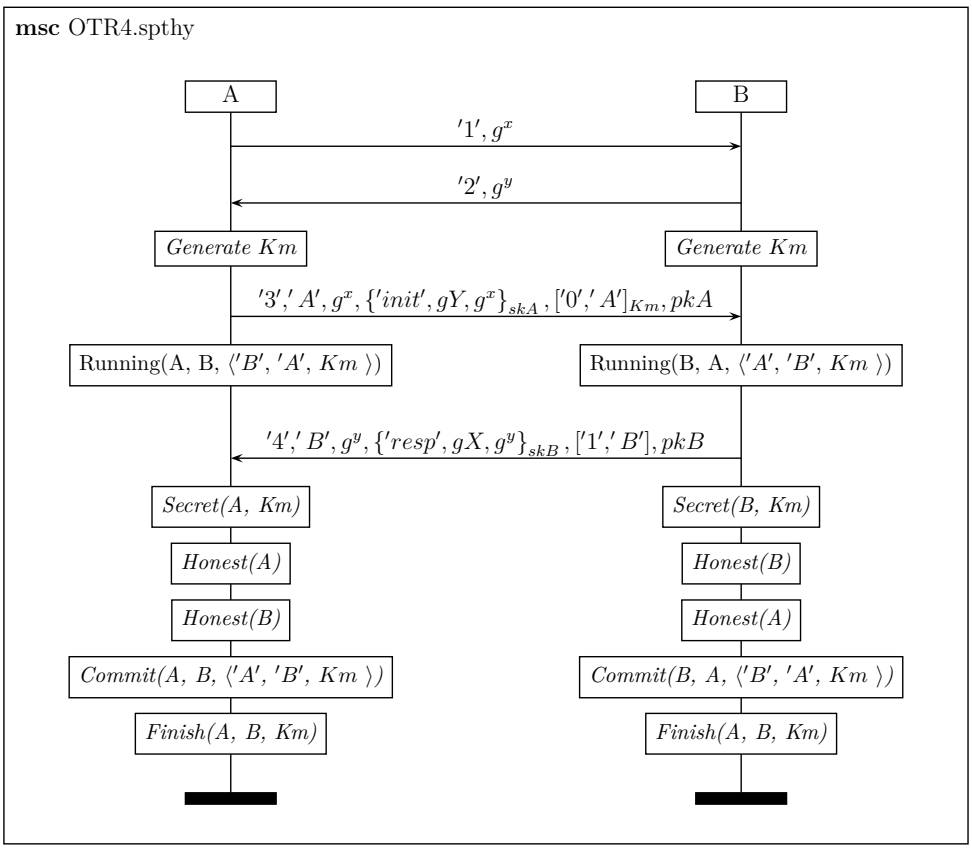

Figure 12: OTR4 specification

The result of running the prover on the formalization of `SIGMA-R` can be seen in `OTR4-proof.spthy`. As for the previous protocol, secrecy (in both forms) of the session key is

provided. Also, this improvement now provides (non-)injective agreement from A's point of view on K_m. However, this protocol does not provide agreement (neither of the two) from the responder's side. The attack found by the tool is fairly simple. A sends the first message to B', but the attacker intercepts it and redirects it to B. The responder answers with its key g^y which is sent to A. Again, the initiator sends the third message to B' but this is forwarded to B. Finally, B sends the last message, which is discarded by the attacker. As depicted in the corresponding figure in `OTR/Proofs`, there is a `Commit` claim by B without a corresponding `Running`. Note that the last message, discarded by the attacker, would let A notice that it is not talking to the intended peer. However, a similar guarantee is not present on B's side, which is why agreement does not hold. The attack, in *Alice&Bob* notation, is shown below.

$A \to I(B') : g^x$
$I(A) \to B : g^x$
$B \to A : g^y$
Generate $K_m = h(g^{xy})$
$A \to I(B') : A, \{g^y, g^x\}_{skA}, [\prime 0', A]_{K_m}\ pkA$
$I(A) \to B : A, \{g^y, g^x\}_{skA}, [\prime 0', A]_{K_m}\ pkA$
$B \to I : B, \{g^x, g^y\}_{skB}, [\prime 1', B]_{K_m}\ pkB$
I discards last message
B commits without a corresponding `Running` *claim*

4 Conclusions

In this project we learnt how to use TAMARIN to formalize protocols and analyse their security properties and guarantees. We saw that this is an important part in designing a protocol since it allows to find attacks which are not always easy to come up with. We also saw that protocols that are used in real world applications are not immune to attacks. Finally, we came across some situations where we could be tricked into thinking that a particular protocol is secure, whereas it is not. In particular, in *Section 2.2* we saw that combining two instances of a simple secure protocol does not guarantee that the resulting one will be secure as well.

References

[1] Ralf Sasse, Cristoph Sprenger. Project Assignment. *Formal Methods For Information Security*, ETH Zürich, FS 2018.

[2] Mario Di Raimondo, Rosario Gennaro, and Hugo Krawczyk. Secure Off-the-Record Messaging. In *WPES*, pages 81–89, 2005

[3] `https://www.bsi.bund.de/EN/Topics/ElectrIDDocuments/SecurityMechanisms/securPACEsecuritymechanismsPACE.html`

[4] `https://en.wikipedia.org/wiki/Off-the-Record_Messaging`