Formal Methods for Security and Privacy (SS21) Prof. Matteo Maffei

Project 1 – Proverif

Due | *22.04.2021* TU Wien

Setting up Your System for the Project

In order to complete this project, you will have to install the tool `proverif`[1]. Please download `proverif` (version 2.02) from

`http://prosecco.gforge.inria.fr/personal/bblanche/proverif/`

and install it on your machine.

You will write the code in the untyped mode like we did in the lecture. Hence, your files should run by executing the command

```
proverif -in pi <your-file>
```

Files that have syntax errors or do not comply to this standard will not be considered. For technical questions, questions to the syntax, semantics, and example code, we refer to the proverif manual, which you can also download from the above url.

Remarks on ProVerif

- Before you start with the project read the submission instructions carefully. Since there will often be hints at the end of the exercises, please read the *whole* exercise before starting it.
- Be neat and hand in readable (possibly annotated) ProVerif files since we will have to manually check the faithfulness of your model. A comment in ProVerif can be written as `(* this is a comment *)`
- Please use the file names specified in the exercise descriptions and the submission instructions.
- Public encryption keys and verification keys for signatures are always public in real life. In order to provide a faithful model, you will have to make them known to the attacker by outputting them on a public channel. For example:

```
free c.
new kA;
!out(c,ek(kA)) | !out(c,vk(kA)) | P
```

models the distribution of Alice's public encryption and verification key over the public network. For the sake of convenience, you might assume that the honest participants have already exchanged their public keys before the protocol starts: so if Bob wants to use Alice's public encryption key he can do so by simply calling `ek(kA)` (he should of course *never* use her secret key), so in the above example `P` of the form

```
let A =
  in(c,x);
  let y = deca(x,dk(kA)) in 0.

let B =
  new n;
  out(c,enca(n,ek(kA))).

let P = A | B.
```

is acceptable.

- Sometimes trace properties, for instance, non-injective agreement, (queries in ProVerif) are trivially fulfilled if events are unreachable. In order to check whether event `Event` is reached in your model, you can use `query ev:Event(x1, ..., xn).`
- Throughout the whole project description, $[\cdot]_{\mathrm{sk}_{kI}}$ refers to the digital signature issued by the principal I. Likewise, $\{\cdot\}_k$ and $\{\cdot\}_{\mathrm{ek}_{kI}}$ refer to symmetric (private-key) encryption with key k and assymmetric (public-key) encryption with I's public encryption key, respectively.
- Most of the protocols in this project contain implicit checks after message arrival. For instance, when the client sends a nonce to the server and expects the same nonce in the server's answer, then this check must be reflected in the ProVerif code; or if a signature arrives that is supposed to be signed by a specific signing key then you should check that this signature verifies using the corresponding verification key.

General Remarks

You will find questions in most of the following exercises. Please make sure to answer all these questions and insert explanations where required in the `project1.pdf` file. We emphasize that it is inherently necessary to answer the questions and fill explanations in order to receive full points.

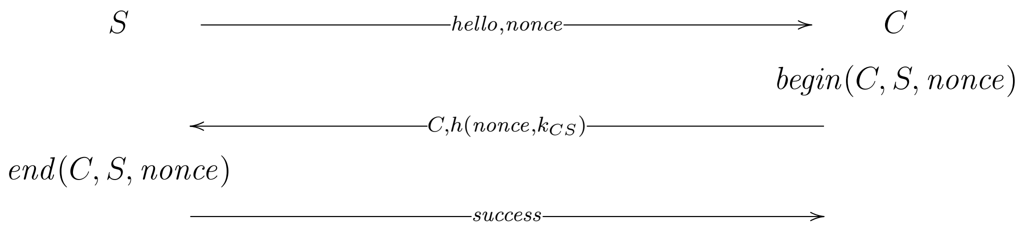
Exercise 1.1: Warm Up (5 points)

Consider the following protocols. For each of them do the following:

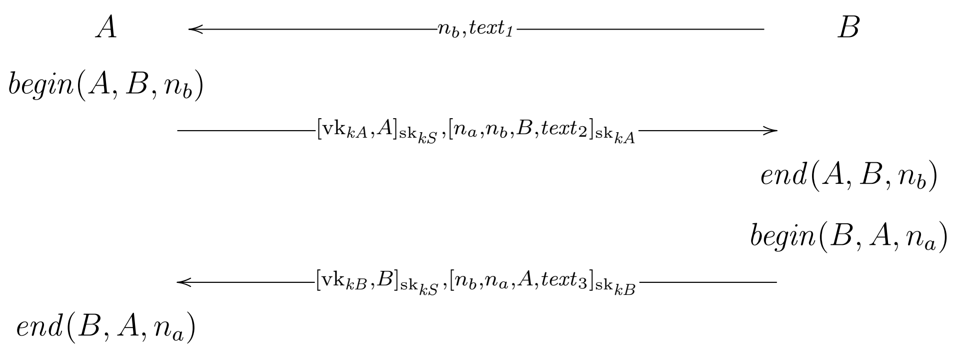
1. model the protocol in ProVerif (call the files `warmup_a.pv`, `warmup_b.pv`, and `warmup_c.pv`);
2. check in each protocol non-injective agreement and injective agreement, respectively;

3. if one or both properties do not hold, describe an attack in the `project1.pdf` file and try to fix the protocol (without changing it dramatically), call the files `filenameMOD.pv` where `filename.pv` is the name of your first ProVerif model.

(a) Here, *hello* and *success* are free names and *nonce* is a fresh nonce. h is a hash function and k_{CS} is a symmetric long-term key shared between C and S.



(b) Here, $[\mathrm{vk}_{kA}, A]_{\mathrm{sk}_{kS}}$ and $[\mathrm{vk}_{kB}, B]_{\mathrm{sk}_{kS}}$ are certificates signed by a trusted server S on the public verification keys of A and B, respectively. The messages $text_i$ are publicly known messages and n_a and n_b are fresh nonces.


A
$n_b, text_1$
B
$begin(A, B, n_b)$
$[\mathrm{vk}_{kA}, A]_{\mathrm{sk}_{kS}}, [n_a, n_b, B, text_2]_{\mathrm{sk}_{kA}}$
$end(A, B, n_b)$
$begin(B, A, n_a)$
$[\mathrm{vk}_{kB}, B]_{\mathrm{sk}_{kS}}, [n_b, n_a, A, text_3]_{\mathrm{sk}_{kB}}$
$end(B, A, n_a)$

Hints

- In order to succeed in this exercise it is not enough to just model the protocol in one direction. Both the left and the right hand-side parties should be able to initiate a communication. Check in the lecture slides how this problem can be solved.
- Note that unless it is leaked, only the trusted server S knows sk_{kS}– not even honest agents. This means that server certificates can not be generated by anyone other than the server.

Exercise 1.2: Self-Signed Certificates (10 points)

Many small private Websites use self-signed certificates. When you visit them, your browser complains that the certificate used by the server is self-signed and should, hence, not be trusted. In this exercise you will investigate where the problems with self-signed certificates lie and how the issue can be fixed. The deliverables for this exercise are two files `self-signed-auth.pv` and `ca-signed-auth.pv`.

To start, consider the following simplified TLS protocol using self-signed certificates:

$$
\begin{array}{ccc}
C & \xrightarrow{\quad \mathit{hello}, n_C \quad} & S \\
 & \xleftarrow{\quad \mathit{hello}, n_S \quad} & \\
 & \xleftarrow{\quad [\mathrm{vk}_S, \mathrm{ek}_S]_{\mathrm{sk}_S}, \mathrm{vk}_S \quad} & \\
 & \xrightarrow{\quad \{\mathit{secret}\}_{\mathrm{ek}_S} \quad} & \\
k = h(\mathit{secret}, n_C, n_S) & & \\
 & \xrightarrow{\quad \{m\}_k \quad} & \\
 & \xleftarrow{\quad \{\mathit{ack}\}_k \quad} &
\end{array}
$$

where *hello* and *ack* are public messages, n_C and n_S are fresh nonces, *secret* is a secret chosen by the client C, and m is a shared secret between the client C and the server S. Note that for this protocol we assume that the server's public key ek_S is *not* known to the client. Perform the following tasks:

1. Model the above protocol in ProVerif and check whether it fulfills non-injective agreement in each direction. To this end, you have to introduce event annotation at appropriate places and issue two queries. If you did it correctly, ProVerif will report *two* different attacks.

2. Describe these two theoretical attacks.

3. Finally, fix the authentication protocol by replacing self-signed certificates with certificates signed by a certification authority. Explain why this fixes the problem.

Hints

1. Unless leaked, only the certification authority should know its own signing key. This means that trusted certificates can not be generated by anyone else.

Exercise 1.3: Commitment Schemes (9 points)

A commitment scheme is a cryptographic primitive that can best be explained intuitively as follows: in order to commit to a message m, one writes m on a piece of paper and puts it in an envelope, seals the envelope, and puts it on top of a table. The message inside cannot be changed and no one can look at it without opening the envelope. These two properties are called *binding* and *hiding*, respectively. More formally, symbolically, a commitment scheme consists of two functions: commit that takes the message m and a key k and produces a commitment c; and open that takes the commitment c, a message m, and a key k, and it succeeds if and only if $c = \mathsf{commit}(m, k)$.

The goal of this exercise is to analyze whether the value to which Alice commits to is strongly secret, that is, an adversary cannot tell a difference between two commitments. To this end, consider the following commitment scheme:

Phase 1: Commit:

$$\mathit{Alice} \xrightarrow{\quad \mathsf{commit}(m,k) \quad} \mathit{Server}$$

Phase 2: Open:

$$
\begin{array}{ccc}
\mathit{Alice} & \xrightarrow{\quad m,k \quad} & \mathit{Server} \\
 & & \text{output } m
\end{array}
$$

where m is the message Alice commits to and k is the commitment key.

ProVerif allows for checking secrecy of a message m up to a certain phase using

```
query attacker:m phase 1.
```

Unfortunately, *strong* secrecy cannot be checked with respect to individual phases.

1. Model the commit phase of the protocol in ProVerif in `commit1.pv`. Check secrecy and strong secrecy of m and write down your results.
2. Model the commit and the open phase of the protocol in ProVerif using phases in `commit2.pv`. Check secrecy of m in phase 1 and secrecy of m in phase 2. Moreover, check strong secrecy of m for the whole protocol. Write down your findings.

In order to commit to more than one message at once, one uses so called vector commitments. Since we can only model a constant number of messages, let us for the moment assume that we want to commit to three messages at once. Consider the following two possibilities to do that:

Phase 1: Commit:

$$\text{Version 1:} \quad \textit{Alice} \xrightarrow{\mathsf{commit}(m_1,m_2,m_3,k_1,k_2,k_3)} \textit{Server}$$

$$\text{Version 2:} \quad \textit{Alice} \xrightarrow{\mathsf{commit}(m_1,m_2,m_3,k)} \textit{Server}$$

Phase 2: Open:

$$\text{Version 1:} \quad \textit{Alice} \xrightarrow{m_2,k_2} \textit{Server}$$

$$\text{output } m_2$$

$$\text{Version 2:} \quad \textit{Alice} \xrightarrow{m_2,k} \textit{Server}$$

$$\text{output } m_2$$

In each version the commitment can be opened for each message individually, that is, in order to open the second message one simply sends m_2 and k_2 (version 1) or k (version 2).

3. Model both protocol versions in ProVerif including both the commit and the open phase. Call the files `version1.pv` and `version2.pv`. Notice that you have to model different commit and open functions for the different versions of commitment schemes.
4. Check again for secrecy and strong secrecy, but this time only for the messages that are not opened, that is, m_1 and m_3.

   If you did everything correctly, both versions preserve the secrecy of all un-opened messages in all phases. Moreover, one version preserves the strong secrecy of the un-opened messages while the other one does not. Explain the reason for that: consider both the symbolic world (ProVerif in particular) and the real world. Which applications can you imagine where the version judged insecure by ProVerif is definitely insecure in the real world? How do the messages have to be?

Exercise 1.4: SAML Single Sign-On (8 points)

On TUWEL you will find the paper *Formal Analysis of SAML 2.0 web browser single sign-on*. It describes how the authors apply formal methods to analyze SAML 2.0, which is a protocol for authenticating a client with a service provider via a third party identity provider. SAML is widely used in the real world, with well-known identity providers such as Google, Microsoft and Facebook. The paper mentioned above shows that a variation of this protocol implemented by Google Apps contained vulnerabilities. Read through the paper, but don't focus too much on the details of the formalization. The main goal of reading is to understand the SAML protocol and the attack that the authors describe in figure 5. After you read the paper, answer the following questions:

(a) In ProVerif, model the vulnerable version of the protocol as described in the paper. Call the file `saml.pv`. The goal of the protocol is that the client is authenticated with the service provider, and that the resource remains secret to the adversary. Model these properties in the file, and make sure that the attack described in the paper is found by the tool.

(b) Modify the protocol you modeled in (a) to prevent the attack. Call the file `saml_MOD.pv`.

Hints

- In the paper it is assumed that communication between honest clients and service providers/identity providers takes place over variations of authentic and confidential channels. You can model this in ProVerif by using the stronger notion of private channels. To model communication with the malignant service provider you can use free channels.
- You can handle authentication of the client with the identity provider via signing.

Exercise 1.5: e-Passports (8 Points)

An e-passport is a passport that comes with an RFID tag, which can store the information printed in the passport as well as additional information like iris scans, fingerprints, and more. e-Passports are used in over 40 countries. Passports use the Basic Access Control Protocol (BAC) to protect the their data. This protocol is given (simplified) below:

$$\begin{tikzcd}
\text{Passport}(k_e, k_m) & \text{Reader}(k_e, k_m) \arrow[l, "\mathsf{GetC}"'] \\
{} \arrow[r, "n_t"] & {} \\
{} & {} \arrow[l, "{[\{n_r, n_t, k_r\}_{k_e}]_{k_m}}"'] \\
{} \arrow[r, "{[\{n_t, n_r, k_t\}_{k_e}]_{k_m}}"] & {}
\end{tikzcd}$$

In the above protocol, the keys k_e and k_m are stored on the passport and scanned by the reader before wireless communication begins. k_m is for MAC verification, which we model for simplicity as symmetric signing/verification.

The IACO e-Passport standard specifies that, when an incorrect or ill-formed message is received, the passport must send out an error code, but it does not specify which. The French implementation is such that if checking a signature fails, `error1` is returned. When a nonce check fails, the protocol returns `error2`.

(a) Remember that *unlinkability* intuitively means that multiple uses of a protocol by the same agent can not be linked to each other. As discussed in the lecture, the French BAC implementation does not guarantee unlinkability. ProVerif can demonstrate this by showing that the process does not adhere observational equivalence. Model the French implementation of the protocol in ProVerif, and make sure that ProVerif outputs a trace that shows the attack. Call your file `passport.pv`.

(b) Fix the protocol such that unlinkability is guaranteed. Call your file `passport_MOD.pv`

Hints

- Create keys for two different passports, and let both interact with the reader first.
- Check for observational equivalence by creating a biprocess.

Submission Instructions

Please submit your solution by *22.04.2021* on TUWEL.

Your submission must be a single archive `your-group-number.zip` containing the following files:

Part	**Required files**
General	`report.pdf`
1.1	`warmup_a.pv`, `warmup_b.pv`, [`warmup_aMOD.pv`, `warmup_bMOD.pv`]
1.2	`self-signed-auth.pv`, `ca-signed-auth.pv`
1.3	`commit1.pv`, `commit2.pv`, `version1.pv`, `version2.pv`
1.4	`saml.pv`, `saml_MOD.pv`
1.5	`passport.pv`, `passport_MOD.pv`

The brackets [ ] around the warm up protocols mean that the files are optional, that is, not necessarily all warm up protocol are to be modified. The `.pdf` file which contains all your answers and explanations should be generated using the `.tex` template provided on TUWEL.

References

[1] B. Blanchet, M. Abadi, and C. Fournet. Automated Verification of Selected Equivalences for Security Protocols. *Journal of Logic and Algebraic Programming*, 75(1):3–51, Feb.–Mar. 2008.