Protocol verification with Proverif

Suela Kodra
University of Tartu, December 2015

1 Introduction

The verification of security protocols have been and is still a very active research area. A common method used to argue about the security of protocols is formal verification. However this method has shown to be error-prone and it also presents some additional difficulties such as the complexity of the proofs. In order to gain more confidence on the security of these protocols, research has been conducted on tools for automatic verification with the formal method.

ProVerif is a known automatic verifier for cryptographic protocols defined in the so-called Dolev-Yao model. This tool verifies the security properties of secrecy, authentication and observational equivalences, under the assumption that the cryptographic primitives are idealized. This means that in the Dolev-Yao model used by ProVerif, an attacker can not learn from the encrypted messages without the possession of the required keys. The tool is capable of attack reconstruction whether a property cannot be proved, an execution trace which falsifies the desired property is constructed.

In this paper, we provide a short introduction on how ProVerif is used, discuss its most important features and have a quick look at how its internals work. The remainder of the paper is organized as follows. In Section 2, we briefly introduce the tool usage and in Section 3, we discuss how it works internally. We conclude the paper in Section 4 by introducing a running example.

2 Using ProVerif

As mentioned before, ProVerif is a tool for automatic verification of security protocols. This tool verifies the protocol for an unbounded number of runs (sessions), using unbounded message space. It has been used and developed since 2001. ProVerif has been successfully used to automatically analyse the security of cryptographic protocols used in electronic voting or key exchange [1]. The tool can be downloaded at **http://prosecco.gforge.inria.fr/personal/bblanche/proverif/.**

In ProVerif, protocols are modelled and verified using the syntax of process calculus of Blanchet et al. [2], which is a type variant of applied π-calculus. The applied π-calculus is based on pi calculus with the addition of a rich term algebra to enable the modelling of cryptographic operations used by security protocols [3]. The tool can be executed by typing the following syntax in the command-line:

./proverif [options] <filename>

where the input file <**filename**> may be encoded in several languages. We are going to focus only on protocols encoded in the typed pi calculus with file extension .pi. Some of the most important options are: **-in** [format] for choosing the type of the input file, **-out** [format] for choosing the

output file format and **-lib** [filename] for the library file (e.g **-lib** crypto). After the execution of ProVerif (if the program terminates), it outputs the proof in the command line.

3 Inside ProVerif

In this section, we briefly review the internals of ProVerif and show how can protocols be modeled with it.

3.1 How it works

A rough visualization of the structure of ProVerif is represented in Fig.1. As shown in the figure, ProVerif takes as input: a *model* of cryptographic protocol (described using the applied π-calculus syntax) and *the security properties* that we want to prove. The security properties are modeled as derivability *queries*. Then the input is automatically translated into a set of *Horn clauses*, which are later subject to a *resolution algorithm*. It is also possible to model the cryptographic protocols using Horn clauses since from the start [4]. The tool relies on the fact that the translation into Horn clauses is an approximation. This approximation is *sound*, meaning that, if an attack exists in the cryptographic protocol, then it also exists in the approximation, but it is also *incomplete*, because if an attack is found in the approximation , it does not guarantee that it also exists in the cryptographic protocol.

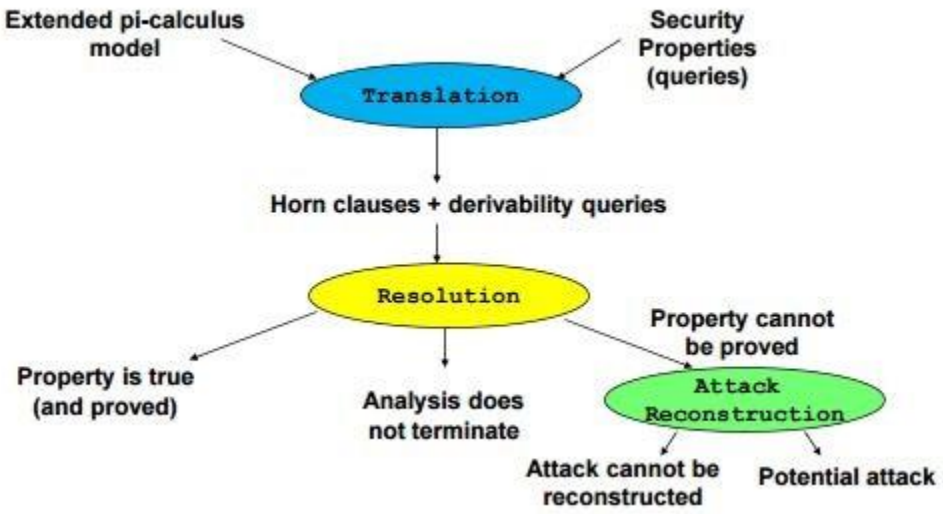

Fig.1. Structure of ProVerif

A Horn clause is a set of predicates, combined by the logical *or* operation, with at most one non-negated literal. It is defined as: $F_1 \wedge ... \wedge F_n \Rightarrow F \ (\equiv \neg F_1 \vee ... \vee \neg F_n \vee F)$, where $n \geq 0$ and F is the only non-negated literal , which is also known as a *fact*. The previous defined Horn clause means that, if all facts F_i *(for i = 1,...,n)* are true, then F is also true. Here, a fact $F = p(M_1, \ldots, M_n)$ shows a property of the messages M_i *(for i = 1,...,n)* and p denotes predicates. The main predicate used in the Horn clauses is ***attacker(M)*** which represents the fact that "the attacker knows the term M". Here, M refers to the messages that are exchanged between protocol´s participants.

We can use this predicate in order to model the actions of the adversary and protocol participants.

These Horn clauses obtained by automatic translation represent the attacker´s potential knowledge, its computational abilities and the protocol itself.

ProVerif executes a so-called *resolution algorithm* using the obtained Horn clauses, in order to verify the derivability of facts. When, the tool verifies that a fact in contradiction to the desired security property can be derived, then it finds an attack. In this case, ProVerif outputs *true* and it also displays some actions that the attacker may take to break the desired security property. However, when a fact in contradiction to the desired security property cannot be derived, a *false* statement is outputted to say that there is no attack. ProVerif does not always terminate and it can also be incomplete (it may generate a *false attack*). The approximation used during the translation into Horn clauses might cause *false attack*, meaning that the derivation of a fact may also correspond to a false attack in the protocol. An infinite loop generated by the Horn clauses might cause non-termination of ProVerif. However, the two last outcomes (non-termination and false attack) of ProVerif rarely happen in practice.

As it is mentioned in the introduction, ProVerif can verify the secrecy and authentication properties. Proving secrecy property is the most basic capability of the tool. To test secrecy of the term *M*, ProVerif attempts to verify that the state in which the term M is known to the adversary is unreachable. Authentication can be defined using correspondence assertions. These are used to capture relationships between events that can be expressed in the form "if some event has been executed, then another event has previously been executed." [1].

3.2 The syntax

A protocol model in ProVerif, with a type variant of applied π-calculus as input language, is organized into three parts: *the declarations, the process macros* and *the main process.*

Declarations. They include the *functions* that describe the cryptographic primitives, the *security properties* and the *user types* (introduced in the updated user manual, but not included in our protocol model). Functions are divided in two categories: *constructors* and *destructor*s. Constructors serve for building terms. Thus, we have the constructor applications of the form: $f(M_1,M_2 \ldots M_k)$. We have presented a general form of both constructor and destructor applications in figure 2. A constructor of arity *n* is defined as ***fun*** *name_of_function/**n.*** On the other hand, the destructor is used to manipulate terms in the processes. So, the destructors are of the form: $h(M_1,M_2 \ldots M_k)$. A destructor *h* of arity k is defined as*:* ***reduc forall*** $x_1,\ldots,x_n$*;* $h(M_1,M_2 \ldots M_k) = M_0$. *Terms* are used to model the messages between participants in the protocol. They are constructed using names, variables, a tuple of terms, constructors, and destructors defined in the declarations. Combined together, constructor and destructor are used to capture the relationship between cryptographic primitives. As typical examples for constructor and destructor, we consider encryption and decryption, respectively. ProVerif uses the keyword **[private]** to declare that free names, constructor, destructor are not known by the attacker.

Processes. The main process is defined using process macros. The grammar for terms and processes of ProVerif input language is given in figure 2. Processes in this grammar are denoted as *P,Q*. The ***null*** *process* does nothing. The replication *!P* defines an unbounded number of copies of process *P* in parallel. The name restriction ***new*** *n:t; P* creates a new name *n* (of type *t*) and then executes *P*. The conditional is defined in terms of ***let***. The input process ***in****(M,x:t);P* inputs a

message on channel *M* and then it runs *P* with variable *x*. The output process ***out****(M,N);P* output message *N* on the channel *M* and then runs *P*. Paper [4] provides more details about this syntax. Each participant of the protocol is modeled as a process. Some auxiliary events are used to specify the security properties without influencing the behavior of the process.

There are three different *query-lines*:

- *query attacker: <message>.* queries the secrecy of the message <*message*>. If the attacker finds a way to learn <message>, the query fails.
- *query ev : <event1> ==> ev : <event2*>. queries whether an occurrence of event <*event1*> implies that <*event2*> has occurred at least once before. For example, authentication between C and S can be seen as a query of the form "if S receives a message m, then C must have sent it before" and later on, prove it using the events.
- *query evinj : <event1> ==> ev : <event2>* . It requires that there are at least as many occurrences of <*event2*> than of <*event1*>.

We usually assume that the processes are executed in the presence of the adversary, which is itself a process in calculus and so it is allowed to execute the same actions as any other process. However, the adversary needs not to be programmed explicitly.

Process macros. The process macros consist of *sub-process* definitions, where each sub-process is a sequence of events. A sub-process P may be defined using macros of the form: ***let*** $R(x_1,\ldots,x_n) = P$, where *R* is the macro name and $x_1,\ldots,x_n$ are free variables of *P*.

We showed in Section 3.1, that the cryptographic protocols modeled using this syntax are automatically later translated into Horn clauses by ProVerif.

M, N ::=	terms
a, b, c, k, m, n, s	names
x, y, z	variables
$(M_1, \ldots, M_k)$	tuple
$h(M_1, \ldots, M_k)$	constructor/destructor application
$M = N$	term equality
$M <> N$	term inequality
not(M)	negation
P, Q ::=	processes
0	null process
$P \mid Q$	parallel composition
$!P$	replication
new $n : t; P$	name restriction
in$(M, x : t); P$	message input
out$(M, N); P$	message output
if M **then** P **else** Q	conditional
let $x = M$ **in** P **else** Q	term evaluation
$R(M_1, \ldots, M_n)$	macro usage
event $e(M_1, \ldots, M_n); P$	events

Fig.2. Syntax of process calculus in ProVerif

4 An example

Protocol description. Our example protocol proceeds as follows. There are two participants: Alice and Bob. Alice generates a fresh nonce and sends it in the hello message to B. Then, B sends a hash of the shared secret key kAB and its nonce to A. Later on, A validates the hash and accepts or declines the authentication of B to A. The session key in our protocol is both the nonce and the shared secret key between A and B, kAB. A sketch of our protocols is represented below:

A → B: (hello; nonce)
B: begin(B,A,nonce)
B → A: nonce; hash(kAB)
B: end(B,A,nonce)

In our model, we declare the events:

- ***event begin(B,A,ts)***, represents the request from B to create a trusted session with A.
- ***event end(B,A,nonce),*** represents the response from A and it also means that A updates the nonce.

Our interest in this model is to verify the secrecy of the used shared key kAB at the end of the protocol and check whether the protocol correctly realizes the authentication of B to A. In order to challenge the adversary, we write the queries syntax, as the following:

1. query evinj:end(x,y,z) ==> evinj:begin(x,y,z).
2. query attacker: kAB.

We use the same ProVerif code in figure 3, but

In figure 3, the query for authentication is specified in Lines 14-15. Process macros for participants A and B are specified in Lines 17–26 and Lines 28–35, respectively. The main process is also specified in Lines 37-40.

The first query (authenticity one) says that in any run of the program, if event end(x,y,z) occurs then, event begin(x,y,z) must have occurred before. The output of the authenticity query is summarized in the following:

```
A trace has been found.
RESULT evinj:end(x_25,y_26,z_27) ==>
evinj:begin(x_25,y_26,z_27) is false.
RESULT (even ev:end(x_304,y_305,z_306) ==>
ev:begin(x_304,y_305,z_306) is false.
)
-- Query not attacker:kAB_14[]
Completing...
Starting query not attacker:kAB_14[]
RESULT not attacker:kAB_14[] is true.
```

The results inform us that authentication of A to B holds but the authentication of B to A does not hold. This happens because of a replay attack: the adversary can replay the message from B to A, which leads several sessions of A to have the same nonce. In order to fix this flaw, we suggest not to send the nonce as a plaintext, but instead hash it with the shared secret key kAB, meaning that we should modify the Line 23. We also modify Line 33. The two modified code lines are:

```
23      let ok = eq(x,(B,h((nonce,kAB)))) in

33      out(c,(B,h((ts,kAB))));
```

We run the ProVerif again and we get the following result:

```
-- Query evinj:end(x_25,y_26,z_27) ==>
evinj:begin(x_25,y_26,z_27)
Completing...
Starting query evinj:end(x_25,y_26,z_27) ==>
evinj:begin(x_25,y_26,z_27)
RESULT evinj:end(x_25,y_26,z_27) ==>
evinj:begin(x_25,y_26,z_27) is true.
-- Query not attacker:kAB_14[]
Completing...
Starting query not attacker:kAB_14[]
RESULT not attacker:kAB_14[] is true.
```

As it is shown from the above result, the fixed protocol provides mutual authentication of two participants A and B.

Now, we are going to test the second query for the secrecy of the shared secret key. We use the same code as in figure 3, but with some added changes shown in the following lines:

```
9   private free kAB.

37  (* Main process *)
38  process
39      !processA | !processB
```

The output of ProVerif for the secrecy query, is summarized as follows:

```
-- Query not attacker:kAB_14[]
Completing...
Starting query not attacker:kAB_14[]
RESULT not attacker:kAB_14[] is true.
```

As it can be seen from the result, ProVerif is testing the query *attacker(kAB)*. But internally ProVerif attempts to show *not attacker(kAB)* and hence RESULT not attacker(KAB) is true, which means that the secrecy of kAB is preserved by the protocol.

```
(* Security Assumptions *)
free c. (* Public channel *)
free B,A. (* Free names *)
reduc eq(x,x) = x.
fun h/1. (* Free names *)
free hello.
free success.


(* Secrecy query *)
query attacker: kAB.

(* Authenticity query *)
query evinj:end(x,y,z) ==> evinj:begin(x,y,z).

let processA =
    new nonce;
    (** Message 1 **)
    out(c,(hello,nonce));
    (** Message 2 **)
    in(c,x);
    let ok =  eq(x,(B,timestamp,h(kAB))) in
    event end(B,A,nonce);
    (** Message 3 **)
    out(c,success).

let processB =
    (** Message 1 **)
    in(c,(=hello,ts));
    event begin(B,A,ts);
    (** Message 2 **)
    out(c,(B,ts,h(kAB)));
    (** Message 3 **)
    in(c,=success).

(* Main process *)
process
    new kAB;  (* secret symmetric key between A and B*)
    !processA | !processB
```

Fig.3. ProVerif code of the protocol

5 Conclusion

In this paper, we introduced a short tutorial on how to use Proverif, the tool for the verification of the security protocol. We also showed how the tool works internally when verifying a protocol, and a running example is given. ProVerif seems to be fairly accessible to new users because of the detailed user manual.

References

[1]M. Christofi**,** A. Gouget, "Formal Verification of the mERA-Based eServices with Trusted Third Party Protocol". **URL:** http://link.springer.com/chapter/10.1007%2F978-3-642-30436-1_25.

[2]B. Blanchet, B. Smyth, and V. Cheval, "ProVerif 1.92: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial".**URL:** http://prosecco.gforge.inria.fr/personal/bblanche/proverif/manual.pdf

[3] S. Delaune, M. Ryan, and B. Smyth, "Automatic verification of privacy properties in the applied pi calculus⋆". **URL:** http://www.lsv.ens-cachan.fr/Projects/anr-avote/PUBLIS/DRS-ifiptm08.pdf

[4]B. Blanchet , "Automatic Verification of Security Protocols in the Symbolic Model: the Verifier ProVerif".**URL:**
http://prosecco.gforge.inria.fr/personal/bblanche/publications/BlanchetFOSAD14.pdf