Generic verification of security protocols

Abdul Sahid Khan, Madhavan Mukund and S. P. Suresh

Chennai Mathematical Institute,
92 G.N. Chetty Road, T.Nagar, Chennai 600 017, India.
E-mail: {sahid,madhavan,spsuresh}@cmi.ac.in

Abstract. Security protocols are notoriously difficult to debug. One approach to the automatic verification of security protocols with a bounded set of agents uses logic programming with analysis and synthesis rules to describe how the attacker gains information and constructs new messages.

We propose a generic approach to verifying security protocols in Spin. The dynamic process creation mechanism of Spin is used to non-deterministically create different combinations of role instantiations. We incorporate the synthesis and analysis features of the logic programming approach to describe how the intruder learns information and replays it back into the system. We formulate a generic "loss of secrecy" property that is flagged whenever the intruder learns private information from an intercepted message. We also describe a simplification of the Dolev-Yao attacker model that suffices to analyze secrecy properties.

1 Introduction

1.1 Background

Security protocols are specifications of communication patterns which are intended to let agents share secrets over a public network. They are required to perform correctly even in the presence of malicious intruders who listen to the message exchanges that happen over the network and also manipulate the system (by blocking or forging messages, for instance). Obvious correctness requirements include secrecy: an intruder cannot read the contents of a message intended for others, and authenticity: if B receives a message that appears to be from agent A and intended for B, then A indeed sent the same message intended for B in the recent past.

The presence of intruders necessitates the use of encrypted communication. A wide variety of cryptographic primitives – some of whose development involves quite sophisticated number theory – is an essential part of the protocol designer's toolkit. But it has been widely acknowledged that even the use of the most perfect cryptographic tools does not always ensure the desired security goals. (See [AN95] for an illuminating account.) This situation arises primarily because of logical flaws in the design of protocols.

Quite often, protocols are designed with features like ease of use, efficiency etc. in mind, in addition to some notion of security. For instance, if every message of a protocol were signed in the sender's name and then encrypted with the receiver's public key, it appears as if a lot of the known security flaws do not occur. But it is not usual for every message of a protocol to be signed. This could either be for reasons of efficiency or because frequent use of certain long-term keys might increase the chance of their being broken using cryptanalysis. Great care needs to be exercised in such situations. The following example protocol highlights some of the important issues nicely. It is based on a protocol designed by Needham and Schroeder [NS78] and is aimed at allowing two agents A and B to exchange two independent, secret numbers. It uses public-key encryption but does not require agents to sign their messages.

$$\begin{array}{l} \text{Msg 1. } A \rightarrow B : \{x, A\}_{pubk_B} \\ \text{Msg 2. } B \rightarrow A : \{x, y\}_{pubk_A} \\ \text{Msg 3. } A \rightarrow B : \{y\}_{pubk_B} \end{array}$$

Here $pubk_A$ and $pubk_B$ are the public keys of A and B, respectively, and $\{x\}_k$ is the notation used to denote x encrypted using key k. In the protocol, x and y are assumed to be newly generated, unguessable (with high probability, of course!), previously unused numbers, also called nonces (nonce stands for "*n*umber *once* used"). In message 2, B includes A's nonce. On seeing it A is assured that B has received message 1, since only B can decrypt the first message and use x in a later message. Similarly on receipt of the third message, B is assured of A's receipt of y.

At the end of a session of the protocol, both A and B share the secrets x and y and both also know that the other agent knows x and y. But it has been shown [Low96] that x and y are not necessarily known *only to A and B*. (Such a property needs to be satisfied if we want to use a combination of x and y as a key shared between A and B, for example.) The attack (called Lowe's attack) is given below:

$$\begin{array}{lcl} \text{Msg } \alpha.1. & A \rightarrow I & : \{x, A\}_{pubk_I} \\ \text{Msg } \beta.1. & (I)A \rightarrow B & : \{x, A\}_{pubk_B} \\ \text{Msg } \beta.2. & B \rightarrow (I)A & : \{x, y\}_{pubk_A} \\ \text{Msg } \alpha.2. & I \rightarrow A & : \{x, y\}_{pubk_A} \\ \text{Msg } \alpha.3. & A \rightarrow I & : \{y\}_{pubk_I} \\ \text{Msg } \beta.3. & (I)A \rightarrow B & : \{y\}_{pubk_B} \end{array}$$

In the above attack, $(I)A \rightarrow B : m$ means that the intruder is sending message m to B in A's name, whereas $A \rightarrow (I)B : m$ means that the intruder is blocking a message sent by A intended for B. The above attack consists of two parallel sessions of the protocol, one (whose messages are labelled with α) involving A as the initiator and I as responder, and the other (whose messages are labelled with β) involving I (in A's name) as the initiator and B as the responder. (This shows that the names A, B, x and y mentioned in the protocol specification are just placeholders or abstract names, which can be concretely instantiated in different ways when the protocol is run. So according to A and B, they have just had a normal protocol session with I and A, respectively. But I knows better!) After the fifth message above, the intruder gets to know y which is the secret generated by B in a session with someone whom B believes to be A. This shows that the protocol does not satisfy the following property: *whenever an agent B engages in a session of the protocol as a responder and B believes that the initiator is A, then the secret generated by B is known only to A and B.* The seriousness of this flaw depends on the kinds of use the protocol is put to. It is worth noting that this attack does not depend on weaknesses of the underlying encryption mechanism (nor even on some keys being guessed by chance). It is also worth noting that this attack on the (simple enough) Needham-Schroeder protocol was discovered seventeen years after the original protocol was proposed. [Low96] also suggests a fix for the protocol:

Msg 1. $A \rightarrow B : \{x, A\}_{pubk_B}$
Msg 2. $B \rightarrow A : \{x, y, B\}_{pubk_A}$
Msg 3. $A \rightarrow B : \{y\}_{pubk_B}$

It is easy to see that the above attack does not happen anymore, but that still doesn't prove that the protocol does not have any vulnerabilities.

The above discussion illustrates the pitfalls in security protocol design, and also highlights the need for a systematic approach to protocol design and analysis. There are two possible approaches:

- Development of a design methodology following which we can always generate provably correct protocols. The work in [AN96], which gives a flavour of the kinds of useful heuristics which improve protocol design, is a step in this direction.
- Development of systematic means of analysing protocols for possible design flaws. The bulk of the work in formal methods for security protocols focuses on this approach. Here again, there are two possibilities:
  - Development of methods for proving the correctness of certain aspects of protocols.
  - Development of systematic methods for finding flaws of those protocols which are actually flawed.

There has been much work in applying *automated theorem proving* ([Pau98] and [Bol97] are some representative papers) and specialised *belief logics* ([BAN90], [AT91], [GNY90], [SC01] is a sampling of the literature) to prove properties of protocols.

In this paper, we describe our experiments with applying the Spin model-checking tool to verify security protocols. The outline of the paper is as follows. In the rest of this section, we discuss the issues which arise in model checking security protocols, and also set our work in the context of other research in this area. In Section 2, we develop a formal model for security protocols. In the next section, we give a high-level description of how we verify security protocols in Spin, using the Needham-Schroeder protocol as a running example. Our approach makes use of some new observations about the properties required of the intruder. We formally justify our assumptions in Section 4. The final section summarizes the work and discusses future directions. In an Appendix, we provide our Promela code for modelling the Needham-Schroeder protocol.

1.2 Model checking security protocols

Much of the literature in formal methods for security protocols is devoted to methods for detecting flaws in protocols using the *model checking* approach – [Low96], [LR97], [MMS97], [Sch96], and [Sch97] is a representative sample. This approach has enjoyed great success in unearthing bugs in many protocols – long after the protocols had been put into use, in some cases. [CJ97] is a good reference for the many attacks which have been uncovered by formal verification tools.

Model checking security protocols is particular challenging. Unlike many other communication protocols, the actual data which is transferred in the message exchanges is also of importance in security protocols. For instance, in the Needham-Schroeder protocol, the nonce which B receives in the third message should be the same as the one it sent in the second message. But that is not all. The names A, B, n which occur in the protocol descriptions are placeholders. During a run of the protocol, they will be instantiated with concrete agent names like `sahid`, `madhavan`, `spsuresh`, etc., and concrete nonces. Further there may be many instantiations of the same roles occurring in one run of a protocol.

Technically speaking, we are dealing with infinite state systems and even simple problems like reachability are undecidable in various general settings. (See [DLMS99], [ALV02], [Sur03], among others, for more details.) Therefore we impose various external bounds, mainly on the number of different sessions or plays (instantiations of roles) that occur in each run of the protocol. But even then, there is a nontrivial amount of data manipulation involved, and the state space can be huge even when we consider a small number of sessions in each run. We attempt to master this complexity by looking at a generic secrecy property, and by observing the intruder structure can be considerably simplified when we search for a violation of this property.

SPIN has been successfully used in model checking security protocols [MS02]. There are also reports of the use of other general purpose model checkers like Murφ to verify security protocols [MMS97]. Many attacks have been successfully discovered and rediscovered by these tools. But the reports in the literature either do not elaborate much on the details of the intruder model, or present an intruder which is designed with a particular attack in mind. Our work is characterised by a very *simple, yet general* intruder model, which helps us discover all breaches of a particular simple kind of *generic* secrecy property efficiently.

In particular, we allow our intruder to listen in on all the messages communicated over all the channels in the network, construct arbitrary messages out of learnt messages (using the message analysis and synthesis rules which are described in a later section), and at any point of time, send a constructed message to the appropriate agent under anyone's identity. But unlike the general Dolev-Yao intruder, we do not allow our intruder to decrypt messages seen earlier using keys learnt later. This considerably simplifies the Promela code for the intruder.

There has been a lot of work based on logic programming that has been reported in the literature ([BP03], [MS01] and [DMTY97], for instance). To each protocol specification is associated a set of Horn clauses whose variables correspond to the variables occurring in the protocol specification. The negation of the desired security property is encoded as a goal formula, and one attempts to derive the formula using the clauses. In the course of a derivation, the variables are instantiated with concrete terms. These roughly correspond to spawning different sessions of the roles of the protocols, with different instantiations for the nonces and other secrets. It is proved that there is a proof of the goal if and only if there is a run of the protocol which constitutes a breach of the security property.

We feel that the model checking approach offers two advantages over the logic programming approach. Firstly, in the logic programming approach, extracting the counterexample run from a proof of the goal is not always a straightforward matter, whereas the SPIN model checker readily provides us with the counterexample run when it detects a violation of the desired property. Secondly, if the model checker doesn't report an error, we know that there is no attack on the protocol within the bounds set on the parameters. On the other hand, with the logic programming approach, there is no easy way of directly bounding the number of sessions in each run of the protocol.

2 A formal model for security protocols

In this section we present a formal model for security protocols, which includes a description of the Dolev-Yao intruder [DY83]. Our presentation is necessarily brief. A more detailed presentation can be found in [RS05].

We start with a (potentially infinite) set of *agents* Ag, which includes the *intruder* I and the others, who are called *honest agents*. We also start with a set of *keys* K which includes long-term public, private, and shared keys, as well as temporary session keys. For every key k, we denote by $\overline{k}$ its *inverse*. Public keys and their corresponding private keys are inverses of each other, while shared keys are their own inverses. We also assume an initial distribution of long-term keys (for example, A has his private key, everyone's public key, and a shared key with everyone else) which is common knowledge. The set of keys known to A initially is denoted K_A. We also assume a countable set of *nonces* N. $\mathcal{T}_0$, the set of *basic terms*, is defined to be $K \cup N \cup Ag$. The set of *information terms* is defined to be

$$\mathcal{T} \quad ::= \quad m \mid (t_1, t_2) \mid \{t\}_k$$

where m ranges over $\mathcal{T}_0$ and k ranges over K. These are the terms used in the message exchanges below. We use the standard notion of subterms of a term to define $ST(t)$ for every term t.

We model communication between agents by *actions*. An action is either a *send action* of the form $A!B{:}(M)t$ or a *receive action* of the form $A?B{:}t$. Here A and B are distinct agents, A is honest; and M denotes the set of nonces and keys occurring in t which have been freshly generated during this (send) action. We define $\mathit{term}(A!B{:}(M)t)$ and $\mathit{term}(A?B{:}t)$ to be t. The agent B is (merely) the intended receiver in $A!B{:}(M)t$ and the purported sender in $A?B{:}t$. As we will see later, every send action is an instantaneous receive by the intruder, and similarly, every receive action is an instantaneous send by the intruder.

Definition 1 *A* **protocol** *is a pair* $\mathsf{Pr} = (\mathsf{C}, \mathsf{R})$ *where* $\mathsf{C} \subseteq \mathcal{T}_0$ *is the set of* constants *of* Pr *(intended to have a fixed interpretation in all runs of* Pr*, unlike fresh nonces and keys); and* R*, the set of* roles *of* Pr*, is a finite nonempty subset of* Ac^+ *each of whose elements is a sequence of* A*-actions for some honest agent* A*.*

The semantics of a protocol is given by the set of all its runs. A run is got by instantiating each role of the protocol in an appropriate manner, and forming admissible interleavings of such instantiations. We present the relevant definitions below.

An *information state* s is a tuple $(s_A)_{A \in Ag}$ where $s_A \subseteq \mathcal{T}$ for each agent A. $\mathcal{S}$ denotes the set of all information states. Given a protocol $\mathsf{Pr} = (\mathsf{C}, \mathsf{R})$, $\mathsf{init}(\mathsf{Pr})$, the *initial state of* Pr is defined to be $(\mathsf{C} \cup K_A)_{A \in Ag}$.

A *substitution* σ is a partial map from $\mathcal{T}_0$ to $\mathcal{T}$ such that for all $A \in Ag$, if $\sigma(A)$ is defined then it belongs to Ag, for all $n \in N$, if $\sigma(n)$ is defined then it belongs to N, and for all $k \in K$, if $\sigma(k)$ is defined then it belongs to K. The definition is generalised to arbitrary terms and actions in the usual manner.

The notion of information state that we use is very rudimentary. In general, a *control state* would include more detail like the number of current sessions each agent is involved in, how far it has progressed in each of them, and so on. For technical ease, we code up these details in the form of *events*. An event of a protocol Pr is a triple (η, σ, lp) such that η is a role of Pr, σ is a substitution, and $1 \leq lp \leq |\eta|$. The set of all events of Pr is denoted $\mathit{Events}(\mathsf{Pr})$. For an event $e = (\eta, \sigma, lp)$ with $\eta = a_1 \cdots a_\ell$, $\mathit{act}(e) \stackrel{\text{def}}{=} \sigma(a_{lp})$ and $\mathit{term}(e) = \mathit{term}(\mathit{act}(e))$. If $lp < |\eta|$ then $(\eta, \sigma, lp) \rightarrow_\ell (\eta, \sigma, lp+1)$. For any event e, $LP(e)$, the *local past* of e, is defined to be the set of all events e' such that $e' \stackrel{+}{\rightarrow}_\ell e$.

We intend a run of a protocol to be an admissible sequence of events. A very important ingredient of the admissibility criterion is the enabling of events given a particular information state. To treat this formally, we need to define how the agents (particularly the intruder) can build new messages from old. This is formalised by the notion synth and analz derivations.

Definition 2 *A* sequent *is of the form* $T \vdash t$ *where* $T \subseteq \mathcal{T}$ *and* $t \in \mathcal{T}$*.*

An analz*-proof (*synth*-proof)* π *of* $T \vdash t$ *is an inverted tree whose nodes are labelled by sequents and connected by one of the* analz*-rules (*synth*-rules) in Figure 1, whose root is labelled* $T \vdash t$*, and whose leaves are labelled by instances of the* Ax_a *rule (*Ax_s *rule). For a set of terms* T*,* $\mathsf{analz}(T)$ *(*$\mathsf{synth}(T)$*) is the set of terms* t *such that there is an* analz*-proof (*synth*-proof) of* $T \vdash t$*. For ease of notation,* $\mathsf{synth}(\mathsf{analz}(T))$ *is denoted by* $\overline{T}$*.*

Definition 3 *The notions of an action enabled at a state and update of a state on an action are defined as follows:*

- $A!B{:}(M)t$ *is* enabled *at* s *iff* $t \in \overline{s_A \cup M}$*.*
- $A?B{:}t$ *is* enabled *at* s *iff* $t \in \overline{s_I}$*.*

$$\frac{}{T \cup \{t\} \vdash t}\ \mathsf{Ax}_a \qquad \frac{}{T \cup \{t\} \vdash t}\ \mathsf{Ax}_s$$

$$\frac{T \vdash (t_1, t_2)}{T \vdash t_i}\ \mathsf{split}_i (i = 1, 2) \qquad \frac{T \vdash t_1 \qquad T \vdash t_2}{T \vdash (t_1, t_2)}\ \mathsf{pair}$$

$$\frac{T \vdash \{t\}_k \qquad T \vdash \overline{k}}{T \vdash t}\ \mathsf{decrypt} \qquad \frac{T \vdash t \qquad T \vdash k}{T \vdash \{t\}_k}\ \mathsf{encrypt}$$

analz-rules　　　synth-rules

Fig. 1. analz and synth rules.

- *update*$(s, A!B{:}(M)t) \stackrel{\text{def}}{=} s'$ *where* $s'_A = s_A \cup M$, $s'_I = s_I \cup \{t\}$, *and for all agents* C *distinct from* A *and* I, $s'_C = s_C$.
- *update*$(s, A?B{:}t) \stackrel{\text{def}}{=} s'$ *where* $s'_A = s_A \cup \{t\}$ *and for all agents* C *distinct from* A, $s'_C = s_C$.

Definition 4 *Given a protocol* Pr *and a sequence* $\xi = e_1 \cdots e_k$ *of events of* Pr, *infstate*(ξ) *is defined to be update*$(\mathsf{init}(\mathsf{Pr}), act(e_1) \cdots act(e_k))$. *An event* e *is said to be* enabled at ξ *iff* $LP(e) \subseteq \{e_1, \ldots, e_k\}$ *and* $act(e)$ *is enabled at infstate*(ξ).

Definition 5 *Given a protocol* Pr, *a sequence* $\xi = e_1 \cdots e_k$ *of events of* Pr *is said to be a* run *of* Pr *iff:*

- *for all* $i : 1 \leq i \leq k$, e_i *is enabled at* $e_1 \cdots e_{i-1}$,
- *for all* $i : 1 \leq i \leq k$, $NT(e_i) \cap ST(\mathsf{init}(\mathsf{Pr})) = \emptyset$, *and for all* $i < j \leq k$, $NT(e_i) \cap NT(e_j) = \emptyset$. *(This is the* unique origination property *of runs.)*

We denote the set of runs of Pr *by* $\mathcal{R}(\mathsf{Pr})$.

In using the model checker, we typically consider runs of Pr which involve a bounded number of instantiations of roles.

Definition 6 *An atomic term* m *is* secret *at a state* s *if* $m \in \mathsf{analz}(s_A) \backslash \mathsf{analz}(s_I)$ *for some* $A \in Ag$ *– it is known to some honest agent but not to the intruder. Given a protocol* Pr, *and a run* ξ *of* Pr, m *is secret at* ξ *if it is secret at infstate*(ξ). *A run is said to be* leaky *if some atomic term* m *is secret at a prefix of* ξ *but not secret at* ξ.

The secrecy problem *is the problem of verifying whether a given protocol* Pr *has a leaky run.*

3 Protocol verification using Spin

The general approach to verifying security protocols in model-checking tools is by now standard, and amounts to two major steps.

– Formalize the protocol.
– Formalize the behaviour of the intruder.

Our aim is to propose a methodology to achieve these two steps in a manner that can be automated, given a reasonable description of the protocol.

3.1 Formalization of the protocol

As we have seen, a protocol can be described as a pattern of messages exchanged between participants playing specified roles. Each role is easily described as a `proctype` in Spin. What is difficult to formalize is the choice in the way these roles are instantiated in order to find flaws in the protocol.

In our approach, each role instantiation corresponds to a fresh instance of the given `proctype`. To account for the fact that the same agent may play multiple roles, when a `proctype` is instantiated we also provide it with an integer identity. The intruder has a fixed identity, 1.

In the `init` process, we construct at least one instance of each proctype. We then, nondeterministically, construct multiple instances of proctypes to model arbitrary configurations. For instance, in our model of the Needham-Schroeder protocol, `procI`, `procA` and `procB` are the proctypes for the roles intruder, role A and role B, respectively. The code in Figure 2 constructs one instance of each proctype and then upto `KEY_MAX` more instances, each of which is either `procA` or `procB`.

```
init {
  byte j = 3;

  run procI(1);  run procA(2);  run procB(3);

  do  /* create more processes nondeterministically */
    :: break;
    :: j++;
       if
         :: (j >= KEY_MAX) -> break;
         :: else -> if
                      :: run procA(j);
                      :: run procB(j);
                    fi;
       fi;
  od;
}
```

Fig. 2. Nondeterministic instantiation of roles

We also have to model nonces. Rather than deal with them symbolically, as in done, for example, in [MS02], we use a shared global integer `used_nonce`.

Whenever a process requires a nonce, it increments this global variable and uses the corresponding value. Since protocols are usually very short and we only use a limited number of instantiations of each role, we can safely define `used_nonce` as `byte` without running out of fresh nonces.

Public keys are implicitly identified with the identity of the agent. Thus, the public key of agent i is just i. Similarly, shared keys can be encoded using a pair of agent identities.

We use an array of `proc_chan` of synchronous SPIN channels to model the actual channels in the system. Each instance of a role with identity i reads messages on channel `proc_chan[i]`. As we shall see, we allow the intruder to read messages on every channel `proc_chan[i]`. We could also use a model in which messages are always routed via the intruder, in which case we need to include the identity of the recipient in each message.

3.2 Formalization of the intruder

The Dolev-Yao intruder model can be modelled by a process that repeatedly performs the following steps:

- Nondeterministically intercept a message on some channel and update its information.
- Nondeterministically generate a message on some channel using known information.

The intruder updates its information using analz rules and generates fresh messages using synth rules. In general, analz rules involve breaking up a message into its constituent parts, storing all the parts and decrypting previously stored messages using newly acquired keys. In the context of the secrecy problem described in the previous section, it suffices to use a simplified version of the Dolev-Yao intruder model in which the intruder never needs to use newly learned keys to decrypt previously stored messages. In effect, the analz phase consists of just breaking up the current message into its constituent parts and decrypting any encrypted component for which the intruder already possesses the decryption key.

Recall that we model nonces and keys using integers. We can thus model the information that the intruder knows using a boolean array indexed by integers (or pairs of integers). Whenever the intruder intercepts a message, the analz rules determine how these arrays are updated.

We also need to record stored messages. One approach would be to have a boolean array indexed by all combinations of message contents, where an entry is `true` whenever the corresponding message has been seen by the intruder. However, since we are looking at a limited number of interleavings of relatively short sequences of messages, it is more efficient to just maintain a list of stored messages in an array.

Generating a message amounts to nondeterministically choosing a recipient, a message type and populating each field in the corresponding message with some known information of the appropriate type. Alternatively, the intruder could simply replay an entire stored message.

3.3 Formalization of the secrecy property

Recall that a secret is formally defined as information that is introduced during the run of the protocol by an honest agent but which is not known to the intruder at the time of its introduction. A secret leaks if it is intercepted by the intruder after having been sent by its originator to some other honest agent in the system.

We could also consider situations in which the secret leaks directly to the intruder. For instance, A could generate a nonce n_A that is sent unencrypted and is intercepted by the intruder. We do not consider this to be an unintended leakage of a secret. It is not difficult to modify our approach to consider such situations also as leakage of secrets.

Since we keep track of the information that the intruder knows, it is quite straightforward to flag an error when the intruder learns new information. To ensure that this meets our definition of when a secret leaks, for each secret nonce (respectively, key) i, we set the boolean `nonce_introduced[i]` (respectively, `key_introduced[i]`) to `true` when i is first received by any agent. In the Needham-Schroeder protocol, we are only interested in loss of secrecy for nonces, so we have the following code to flag loss of secrecy. This code is invoked whenever the intruder sets `known_nonce[k]` to `true`.

```
inline modifyFlawStatus(k) {
  if
    :: (nonce_introduced[k] == true) ->
       flaw = true;
    :: else -> skip;
  fi;
}
```

We can then write a generic verification condition for loss of secrecy in LTL as `[](!flaw)`.

3.4 Some experimental results

The Appendix describes our Promela code for verifying the Needham-Schroeder protocol. When we ran the code through SPIN, it discovered Lowe's attack, as shown in Figure 3. Notice that this counterexample only uses the basic 3 processes created initially, even though the Promela code permits the creation of additional role instantiations.

We also ran a modified version of this code in which the system necessarily generated 3 instantiations each of roles A and B. Interestingly, the counterexample reported by SPIN corresponds to Lowe's attack involving the intruder and the last two copies of A and B, with some irrelevant intervening messages between the other instances of A and B, as shown in Figure 4. When SPIN was asked to find the shortest counterexample, it found the version of Lowe's attack involving just the first three processes.

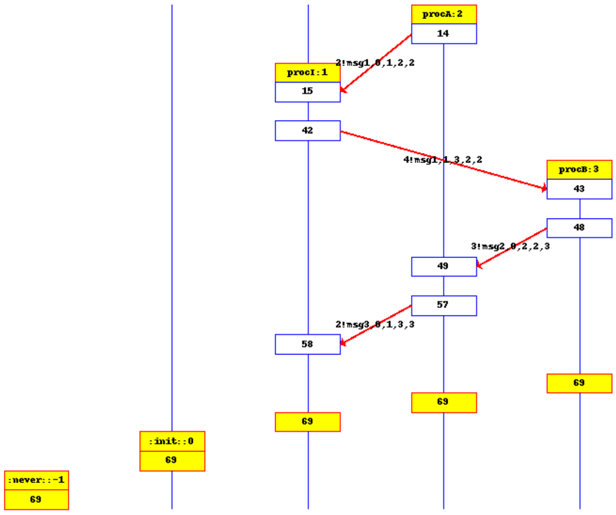

Fig. 3. Lowe's attack on the Needham-Schroeder protocol, discovered by SPIN

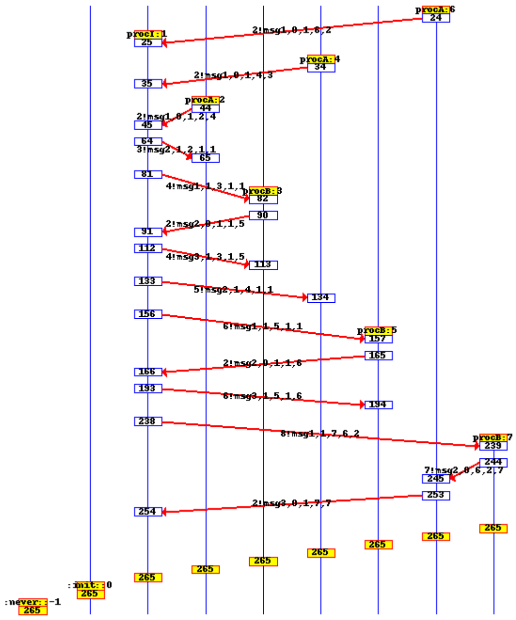

Fig. 4. Lowe's attack with 6 processes

3.5 Comparison with earlier approaches

Ours is not the first attempt to use Spin to verify protocols. Another attempt is described in [MS02]. In the earlier approach, instead of using synth and analz rules to describe the behaviour of the intruder, the intruder is allowed to generate arbitrary messages. Static analysis is used to limit the intruder's choices to "useful" messages. Nonces and other data are handled using symbolic names. Finally, the security property is formulated explicitly, keeping in mind the nature of the protocol. Though the authors claim that their approach can be automated, all of these factors appear to indicate the need for manual analysis before invoking Spin.

In contrast, our approach formulates the intruder and the security property in a sufficiently generic manner that the Promela code for verifying a protocol can, in principle, be synthesized automatically from a suitable description of the protocol, using a system such as CAPSL [DMR00] or Casper [Low98].

4 A simpler intruder model for secrecy properties

We now formalize the simplified intruder used in our Promela model and prove that it is as powerful as the Dolev-Yao intruder, as far as violating the secrecy property of Definition 6 is concerned.

In what follows, for ease of notation, we fix a protocol Pr which has a leaky run, and fix a leaky run $\xi = e_1 \cdots e_k$ of Pr of shortest length. For all $i : 1 \leq i \leq k$, we let t_i, ξ_i and T_i denote $term(e_i)$, $e_1 \cdots e_i$ and $(infstate(\xi_i))_I$, respectively. We let T_0 denote $\mathsf{C} \cup K_I$. For all $i \leq k$, define T_i' and N_i by induction as follows: $T_0' = N_0 = \mathsf{C} \cup K_I$, $T_{i+1}' = T_i' \cup \mathsf{analz}(\{t_{i+1}\} \cup N_i)$ and $N_{i+1} = T_{i+1}' \cap \mathcal{T}_0$. In other words, after every event e_i, the intruder stores the nonces and keys known till now in N_i. As soon as e_{i+1} is seen, the keys from N_i are used to decrypt t_{i+1} and learn more nonces and keys.

Lemma 7 *There is some atomic term n_0 which is secret at ξ_{k-1} and which belongs to N_k. Further, for all $i \leq k$, if e_i is a receive event then $t_i \in \mathsf{synth}(T_i')$.*

Proof. Since ξ is a leaky run of shortest length, none of its proper prefixes is a leaky run and hence it is clear that for all $i < k$, ξ_i is not leaky, which means that none of the terms known to the intruder at the end of ξ_i is a secret at ξ_j for any $j < i$. In other words, for all $i : 1 \leq i < k$ and for all $m \in \mathsf{analz}(T_i) \setminus \mathsf{analz}(T_{i-1})$, $m \notin \mathsf{C}$, $m \notin K_A$ for any $A \in Ag$, and $m \notin ST(t_j)$ for any $j < i$.

Let m_0 be a secret at ξ_{k-1} which belongs to $\mathsf{analz}(T_k)$ (and, of course, does not belong to $\mathsf{analz}(T_{k-1})$). Let π be an analz-proof of $T_k \vdash m_0$, such that for all subproofs π_1 of π, if $T_k \vdash t$ labels the root of π_1 and $t \in \mathsf{analz}(T_i) \setminus \mathsf{analz}(T_{i-1})$ for some $i \leq k$, then all the leaves of π_1 are labelled only by terms t_j for $j \leq i$. Let ϖ be a subproof of π whose root is labelled $T_i \vdash n_0$ for some n_0 which is secret at ξ_{k-1}, and such that none of the terms labelling the nonroot nodes of ϖ is a secret at ξ_{k-1}.

We now prove by induction on the structure of ϖ the claim that for all terms t labelling a node of ϖ, $t \in T'_i$ for the least i such that $t \in \mathsf{analz}(T_i)$. Once we prove this claim, it immediately follows that $n_0 \in T'_k$ and, since $n_0 \in \mathcal{T}_0$, $n_0 \in N_k$. Further let e_i be a receive event for some $i \leq k$. Then from the admissibility conditions it is clear that $t_i \in \overline{T_{i-1}} = \mathsf{synth}(\mathsf{analz}(T_{i-1}))$. But it is an easy consequence of the claim that $\mathsf{analz}(T_j) \subseteq T'_j$ for all $j < k$, and it immediately follows from this that $t_i \in \mathsf{synth}(T'_{i-1})$.

We now turn to the proof of the claim. There are three cases to consider.

- The base case is when t labels a leaf node of ϖ. But this means that there is some $i \leq k$ such that $t = t_i$. By our assumption on π, it is clear that $t \notin \mathsf{analz}(T_{i-1})$. Therefore i is the least number such that $t \in \mathsf{analz}(T_i)$. Clearly enough, $t \in T'_i$ as well.
- The other case is when t labels the conclusion of a split rule. Let i be the least number such that $t \in \mathsf{analz}(T_i)$. Let t' label the premise of the rule. Let i' be the least number such that $t' \in \mathsf{analz}(T_{i'})$. By our assumptions on π, it is clear that $i' \leq i$. By the induction hypothesis $t' \in T'_{i'} \subseteq T'_i$. From this another application of the split rule tells us that $t \in T'_i$.
- The most interesting case is when t labels a conclusion of a $\mathsf{decrypt}$ rule. Let i be the least number such that $t \in \mathsf{analz}(T_i)$. Clearly $\{t\}_k$ and $\overline{k}$ label the premises of the rule, for some key k. Let i' and i'' be the least numbers such that $\{t\}_k \in \mathsf{analz}(T_{i'})$ and $\overline{k} \in \mathsf{analz}(T_{i''})$. By our assumptions on π, it is clear that $i' \leq i$ and $i'' \leq i$.
  Now it cannot be the case that $i' < i''$. This is because, letting j be the least number such that k occurs as a subterm of t_j, it is easy to see that $j \leq i'$, and $\{k, \overline{k}\} \subseteq \mathsf{analz}((\mathit{infstate}(\xi_j))_A)$ for some $A \in \mathit{Ho}$. Now, since i'' is the least number such that $\overline{k} \in \mathsf{analz}(T_{i''})$, this means that $\overline{k}$ is a secret at ξ_j, which is a contradiction because it labels a nonroot node of ϖ. Therefore $i'' \leq i'$.
  By the induction hypothesis $\{t\}_k \in T'_{i'}$ and $\overline{k} \in T'_{i''}$. If $i'' < i'$, then $\overline{k}$ would belong to $T'_{i'-1}$ and hence to $N_{i'-1}$ (since $\overline{k} \in \mathcal{T}_0$). Then it is easy to see that $t \in T'_{i'} \subseteq T'_i$, as desired. On the other hand, if $i' = i''$ then $\{\{t\}_k, \overline{k}\} \subseteq \mathsf{analz}(\{t_{i'}\} \cup N_{i'-1})$, and hence $t \in \mathsf{analz}(\{t_{i'}\} \cup N_{i'-1}) \subseteq T'_{i'} \subseteq T'_i$, as desired.

The above lemma shows that whenever a Dolev-Yao intruder captures a secret, so does an intruder that does not decrypt earlier messages using keys it has learnt later. This justifies the intruder model of Section 3.

5 Conclusion

We have described an approach for generic verification of secrecy properties of security protocols using Spin. For some protocols, correctness is described in terms of authentication rather than secrecy. We do not yet have a uniform method for describing authentication properities in our framework. We have also not yet embarked on the ambitious programme of writing a compiler from a specification language such as CAPSL into Spin to automatically generate verification models for arbitrary protocols.

References

[ALV02] Roberto M. Amadio, Denis Lugiez, and Vincent Vanackère. On the symbolic reduction of processes with cryptographic functions. *Theoretical Computer Science*, 290(1):695–740, 2002.

[AN95] Ross Anderson and Roger M. Needham. Programming Satan's computer. In *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 426–441, 1995.

[AN96] Martin Abadi and Roger M. Needham. Prudent engineering practices for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22:6–15, 1996.

[AT91] Martin Abadi and Mark Tuttle. A Semantics fo a Logic of Authentication. In *Proceedings of the 10th ACM Annual Symposium on Principles of Distributed Computing*, pages 201–216, Aug 1991.

[BAN90] Michael Burrows, Martin Abadi, and Roger M. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, Feb 1990.

[Bol97] Dominique Bolignano. Towards a mechanization of cryptographic protocol verification. In *Proceedings of CAV'97*, volume 1254 of *Lecture Notes in Computer Science*, pages 131–142, 1997.

[BP03] Bruno Blanchet and Andreas Podelski. Verification of Cryptographic Protocols: Tagging Enforces Termination. In Andrew D. Gordon, editor, *Proceedings of FoSSaCS'03*, volume 2620 of *Lecture Notes in Computer Science*, pages 136–152, 2003.

[CJ97] John Clark and Jeremy Jacob. A survey of authentication protocol literature. Electronic version available at `http://www.cs.york.ac.uk./~jac`, 1997.

[DLMS99] Nancy A. Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. The undecidability of bounded security protocols. In *Proceedings of the Workshop on Formal Methods and Security Protocols (FMSP'99)*, 1999.

[DMR00] G. Denker, J. Millen, and H. Ruess. The CAPSL Integrated Protocol Environment. Technical Report SRI-CSL-2000-02, SRI, October 2000. Available at `http://www.csl.sri.com/users/millen/capsl`.

[DMTY97] Mourad Debbabi, Mohamed Mejri, Nadia Tawbi, and Imed Yahmadi. Formal automatic verification of authentication protocols. In *Proceedings of the First IEEE International Conference on Formal Engineering Methods (ICFEM97)*. IEEE Press, 1997.

[DY83] Danny Dolev and Andrew Yao. On the Security of public-key protocols. *IEEE Transactions on Information Theory*, 29:198–208, 1983.

[GNY90] Li Gong, Roger Needham, and Raphael Yahalom. Reasoning About Belief in Cryptographic Protocols. In Deborah Cooper and Teresa Lunt, editors, *Proceedings 1990 IEEE Symposium on Research in Security and Privacy*, pages 234–248. IEEE Computer Society, 1990.

[Low96] Gavin Lowe. Breaking and fixing the Needham-Schroeder public key protocol using FDR. In *Proceedings of TACAS'96*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166, 1996.

[Low98] Gavin Lowe. Casper: A compiler for the analysis of security protocols. *Journal of computer security*, 6:53–84, 1998.

[LR97] Gavin Lowe and Bill Roscoe. Using CSP to detect errors in the TMN protocol. *IEEE Transactions of Software Engineering*, 23(10):659–669, 1997.

[MMS97] John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using Murφ. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 141–153, 1997.

[MS01] Jonathan K. Millen and Vitaly Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *ACM Conference on Computer and Communications Security*, pages 166–175, 2001.

[MS02] P. Maggi and R. Sisto. Using SPIN to Verify Security Protocols. In *Proceedings of the 9th International SPIN Workshop on Model Checking of Software*, number 2318 in Lecture Notes in Computer Science, pages 187–204, 2002.

[NS78] Roger M. Needham and Michael D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21(12):993–999, 1978.

[Pau98] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of computer security*, 6:85–128, 1998.

[RS05] R. Ramanujam and S.P.Suresh. Decidability of context-explicit security protocols. *Journal of Computer Security*, 13(1):135–165, 2005.

[SC01] Paul F. Syverson and Iliano Cervesato. The logic of authentication protocols. In Ricardo Focardi and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *Lecture Notes in Computer Science*, pages 63–106, 2001.

[Sch96] Steve Schneider. Security properties and CSP. In *Proceedings of the IEEE Computer Society Symposium on Security and Privacy*, 1996.

[Sch97] Steve Schneider. Verifying authentication protocols with CSP. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, 1997.

[Sur03] S.P. Suresh. *Foundations of Security Protocol Analysis*. PhD thesis, The Institute of Mathematical Sciences, Chennai, India, November 2003. Madras University. Available at `http://www.cmi.ac.in/~spsuresh`.

A Promela model of the Needham-Schroeder protocol

```
#define NONCE_MAX 8
#define KEY_MAX 8
#define STORE_MAX 10

mtype {msg1, msg2, msg3};
byte used_nonce = 1, used_key = 1, s_ptr;

typedef Crypt {
  byte key;
  byte info1;
  byte info2;
};

typedef stored_data {
  mtype msg_type;
  Crypt data;
}
stored_data stored_mesg[STORE_MAX];

typedef channel {
  chan C = [0] of { mtype,  /* message type */
                    bool,   /* is it sent by intruder?  */
                    Crypt };
};
channel proc_chan[KEY_MAX];

bool known_nonce[NONCE_MAX], known_key[KEY_MAX];
bool nonce_introduced[NONCE_MAX];

bool flaw;  /* security property */

/* nondeterministically chooses a known nonce */
inline chooseNonce(nonce) {
  nonce = 1;
  do
    :: (known_nonce[nonce] == true) ->
       break;
    :: nonce++;
       if
         :: (nonce >= NONCE_MAX) ->
            nonce = 1; /* to ensure nonce has a sensible value */
            break;
         :: else -> skip;
       fi;
```

```
  od;
}

/* nondeterministically chooses a running process */
inline getAValidProcess(i) {
  i = 1;
  do  /* choose the process to send the message */
    :: break;
    :: i++;
       if
         :: (i == KEY_MAX - 1) ->
            break;
         :: else -> skip;
       fi;
  od;
}

inline modifyFlawStatus(k) {
  if
    :: (nonce_introduced[k] == true) ->
       flaw = true;
    :: else -> skip;
  fi;
}

inline replayMessage() {
  i = 0;
  do
    :: (i > s_ptr) -> break;
    :: else ->
       if
         :: (1) ->
            msg = stored_mesg[i].msg_type;
            data.key = stored_mesg[i].data.key;
            data.info1 = stored_mesg[i].data.info1;
            data.info2 = stored_mesg[i].data.info2;

            getAValidProcess(i);

            proc_chan[i].C!msg(true, data);
            break;
         :: skip;
       fi;
       i++;
  od;
```

```
}


init {

  byte j;

  run procI(1);
  run procA(2);
  run procB(3);

  j = 3;
  do  /* create processes nondeterministically */
    :: break;
    :: j++;
       if
         :: (j >= KEY_MAX) ->
            break;
         :: else ->
            if
              :: run procA(j);
              :: run procB(j);
            fi;
       fi;
  od;

}

proctype procA(int agent_id) {

  byte my_nonce, recvd_nonce;
  byte i = 1;
  chan my_chan = proc_chan[_pid].C;
  chan p_chan;
  Crypt data;

  atomic {

    getAValidProcess(i);  /* choose the partner */

    p_chan = proc_chan[i].C;
    used_nonce++;
    my_nonce = used_nonce;
    data.key = i; data.info1 = _pid; data.info2 = my_nonce;
    p_chan!msg1(false, data); /* 0 to signify that it is not sent by intruder proc */
```

```
  }

  my_chan?msg2(_, data);

  do  /* if unexpected message go to infinite loop */
    :: ((data.key == _pid) && (data.info1 == my_nonce)) -> break;
  od;

  atomic {
    recvd_nonce = data.info2;
    nonce_introduced[recvd_nonce] = true;
    data.key = i; data.info1 = recvd_nonce;
    p_chan!msg3(false, data);
  }

}

/* Process B */
proctype procB(int agent_id) {

  byte partner, p_nonce, my_nonce;
  chan my_chan = proc_chan[_pid].C;
  chan p_chan;
  Crypt data;

  my_chan?msg1(_, data);
  do /* if unexpected message go to infinite loop */
    :: (data.key == _pid) -> break;
  od;

  atomic {
    partner = data.info1; p_nonce = data.info2;
    nonce_introduced[p_nonce] = true;
    p_chan = proc_chan[partner].C;
    used_nonce++;
    my_nonce = used_nonce;
    data.key = partner; data.info1 = p_nonce; data.info2 = my_nonce;
    p_chan!msg2(false, data);
  }


  my_chan?msg3(_, data);
  do /* if unexpected message go to infinite loop */
    :: ((data.key == _pid) && (data.info1 == my_nonce)) -> break;
  od;
```

```
}

/* Intruder Process */
proctype procI(int agent_id) {

  mtype msg;
  chan temp_chan;
  Crypt data;
  byte i, j, k, nonce;

  atomic {
    known_nonce[1] = true; /* nonce 1 is for the intruder */
    known_key[1] = true;
  }

  do
    ::  /* intercept a message */
       if  /* choose a channel to intercept */
         :: proc_chan[1].C?msg(false, data);
         :: proc_chan[2].C?msg(false, data);
         :: proc_chan[3].C?msg(false, data);
         :: proc_chan[4].C?msg(false, data);
         :: proc_chan[5].C?msg(false, data);
         :: proc_chan[6].C?msg(false, data);
         :: proc_chan[7].C?msg(false, data);
       fi;
       atomic {
         if
           :: (s_ptr < STORE_MAX) ->
              stored_mesg[s_ptr].msg_type = msg;
              stored_mesg[s_ptr].data.key = data.key;
              stored_mesg[s_ptr].data.info1 = data.info1;
              stored_mesg[s_ptr].data.info2 = data.info2;
              s_ptr++;
           :: else -> skip;
         fi;

         if
           :: (msg == msg1) ->
              j = data.info1; k = data.info2;

              if /* if possible decrypt */
                :: (known_key[data.key]) ->
                   if
                     :: (!known_nonce[k]) -> modifyFlawStatus(k);
```

```
          :: else -> skip;
        fi;
        known_nonce[k] = true;

      :: else -> skip;
                nonce_introduced[k] = true;
   fi;

:: (msg == msg2) ->
   j = data.info1; k = data.info2;

   if /* if possible decrypt */
     :: (known_key[data.key]) ->

        if
          :: (!known_nonce[j]) ->
             modifyFlawStatus(i, j);
          :: else -> skip;
        fi;
        known_nonce[j] = true;

        if
          :: (!known_nonce[k]) ->
             modifyFlawStatus(i, k);

          :: else -> skip;
        fi;
        known_nonce[k] = true;

     :: else -> skip;

                nonce_introduced[j] = true;
                nonce_introduced[k] = true;
   fi;

:: (msg == msg3) ->
   j = data.info1;

   if                                  /* if possible decrypt */
     :: (known_key[data.key]) ->

        if
          :: (!known_nonce[j]) ->
             modifyFlawStatus(i, j);
```

```
                   :: else -> skip;
                 fi;
                 known_nonce[j] = true;

              :: else -> skip;
                        nonce_introduced[j] = true;

            fi;

         :: else -> skip;
       fi;

     }
  ::      /* message replay */
     atomic { replayMessage() };

  :: atomic { /* synthesize a message */

    getAValidProcess(i); /* choose the process to send the message */
    temp_chan = proc_chan[i].C; data.key = i;

    if
      :: /* create a message of type 1 */
         chooseNonce(nonce);   /* choose the nonce for 1st field */
         getAValidProcess(i);  /* choose the initiator */
         data.info1 = i; data.info2 = nonce;
         temp_chan!msg1(true, data);

      :: /* create a message of type 2 */
         chooseNonce(nonce);   /* choose the nonce for 1st field */
         data.info1 = nonce;
         chooseNonce(nonce);   /* choose the nonce for 2nd field */
         data.info2 = nonce;
         temp_chan!msg2(true, data);

      :: /* create a message of type 3 */
         chooseNonce(nonce);   /* choose the nonce for 1st field */
         data.info1 = nonce;
         temp_chan!msg3(true, data);
    fi;
  }

 od;
}
```

```
never { /* !([](!flaw)) */
T0_init:
  if
    :: ((flaw)) -> goto accept_all
    :: (1) -> goto T0_init
  fi;
accept_all:
  skip
}
```