Generic and Efficient Attacker Models in SPIN

Noomene Ben Henda
Ericsson Research Stockholm, Färögatan 6 16480, SWEDEN
noamen.ben.henda@ericsson.com

ABSTRACT
In telecommunication networks, it is common that security protocol procedures rely on context information and other parameters of the global system state. Current security verification tools are well suited for analyzing protocols in isolation and it is not clear how they can be used for protocols intended to be run in more "dynamic" settings. Think of protocol procedures sharing parameters, arbitrarily interleaved or used as building blocks in more complex compound procedures. SPIN is a well established general purpose verification tool that has good support for modeling such systems. In contrast to specialized tools, SPIN lacks support for cryptographic primitives and intruder model which are necessary for checking security properties. We consider a special class of security protocols that fit well in the SPIN framework. Our modeling method is systematic, generic and efficient enough so that SPIN could find all the expected attacks on several of the classical key distribution protocols.

Categories and Subject Descriptors
C.2.2 [**Computer-communication Networks**]: Network Protocols—*Protocol verification*

General Terms
Theory, Security, Verification, Experimentation

1. INTRODUCTION

Background. In systems like telecommunication networks, protocol procedures are usually used as building blocks in more complex compound procedures. For example a handover relies on several procedures from different protocols in order to establish a new data path through another base station [1] (TS. 36.300). These comprise the handover request negotiation, the new bearer establishment, the security activation for the new radio connection, etc. Furthermore, the procedures involve several nodes in the network, namely: a terminal, two base stations, a special node for mobility management and a gateway node for the user plane traffic. Adding to the complex nature of such operation, there are strict efficiency requirements in order to make the transition as smooth as possible and thus minimally disrupt the user experience. Such requirements imply various restrictions on the design of the security related procedures which typically have to be integrated in existing communication protocols. For example a common practice in key establishment procedures is the use of counters instead of *nonces* in the session key derivation process. Examples include, using message sequence numbers from underlying transport protocols, or dedicated counters keeping track of the number of runs of some procedure. The problems with counters is that they need to be synchronized among the communicating peers, and that they require additional special procedures for handling situations like when the counter is reset. This is required in order to prevent key re-use.

Formal verification of security protocols has been ongoing for two decades. State of the art tools like Scyther [9], ProVerif [6] and Tamarin [22] can provide unbounded verification. That is, they prove the security properties for unbounded number of agents or sessions. Scyther has been used in different case studies from which we cite [10, 11]. Scyther is user friendly and has a simple modeling language. However, the language does not provide support for state variables and control flow such as loops and conditionals. ProVerif is also widely used [4, 16, 25]. Its modeling language is a typed variant of the pi calculus [5]. It is more expressive than Scyther's providing for example support for conditionals. Compared to the previous tools, Tamarin is the most recent one. It has a low level language that is expressive enough for modeling state variables and counters. Using Tamarin requires advanced knowledge and in general the tool doesn't scale well to complicated models exploiting the full potential of its input language. SPIN [15] uses the Promela language which is a good alternative for its simplicity and expressiveness. However, in contrast to specialized tools, SPIN neither has support for cryptographic primitives nor for an intruder model. Such features are required for checking security properties.

In general, formal analysis of security protocols is performed in the symbolic Dolev-Yao intruder model [13]. In this model, the intruder has full control over the communication medium. Cryptography is assumed perfect so that the intruder cannot decrypt or encrypt messages without the necessary keys. In particular, the intruder has the ability to intercept, record and replay any exchanged messages. Technically, this model induces an infinite state system and thus even simple problems like reachability are undecidable [14]. For that, one has to restrict the intruder model by for example imposing memory bounds. As a matter of fact, the attacks on the protocol examples that we consider require very little of the attacker memory.

We are interested in using SPIN, which is a finite state model-checker, for analyzing security protocols. In particular, we want to analyze to which extent the attacker model can be weakened and still remain efficient, i.e. be able to find real attacks. For that we need to model bounded instances of the protocols. This can be achieved by for example restricting the number of agents running the protocols and using a finite-memory attacker model. In general, even in such finite settings, the problem is still difficult (NP-complete) [24].

Contribution. We consider a simple class of security protocols and we give a precise formalization of the corresponding operational semantics using transition systems. This class of security protocols fits well in the SPIN framework. We illustrate on an example a systematic method for modeling such protocols in Promela in order to check secrecy and authentication properties. We describe and use a weak generic intruder model that is efficient enough to find attacks in several of the well-known broken key distribution protocols.

Related Work. The most relevant works that are similar to ours are [21, 17]. In [21], the authors describe a similar approach for modeling security protocols and illustrate it on the reduced version of Needham-Shroeder (NS) protocol. The approach is different in few aspects such as in the use of several channels for handling different types of messages. In our method, we use one channel and we assume a fixed "one-size-fits-all" message length and we defer the type checking of the message fields to the processes implementing the different roles of the protocol. One other difference is that the method of [21] relies on static analysis to list all possible messages that can be generated by the intruder. The intruder process is then implemented based on that information. Compared to this, our intruder process is generic and can be reused as is for different protocol models. In [17], the authors present a generic method for modeling security protocols that is tailored for checking only secrecy properties. They also illustrate their method on the reduced version of NS. Compared to this our method allows checking authentication as well.

Outline. In the next section, we describe a restricted class of security protocols and formalize the corresponding operational semantics. We dedicate Section 3 to illustrate with an example a systematic method for modeling such class of security protocols in Promela. In Section 4, we present our experimental results with SPIN. We conclude in Section 5 by a summary and future works. For shortage of space, we have left out the full versions of our Promela models. They can be provided on request.

2. SECURITY PROTOCOLS

2.1 Preliminaries and Notations

Security protocols are communication protocols that rely on cryptographic primitives in order to guaranty properties such as *secrecy* of the exchanged messages and *authentication* of the communicating *agents* (or principals). A security protocol can be defined by a list of protocol *rules* of the form of (1). This particular rule represents an agent A sending to another agent B the message x encrypted with the public *key* of B denoted by $pk(B)$.

$$A \longrightarrow B : \{x\}_{pk(B)} \tag{1}$$

Alternatively, a protocol can be defined as a set of *roles* where each role is a non-empty indexed list of *actions*. There are receive and send message actions denoted by `recv` and `send` respectively. Using this notation, the protocol rule of (1) can be expressed as follows:

$$\begin{array}{ll} RoleA: & RoleB: \\ \quad 1.\ \texttt{send}(\{x\}_{pk(B)}) & \quad 1.\ \texttt{recv}(\{x\}_{pk(B)}) \end{array}$$

In general protocol descriptions contain components like variables (A, B), functions (pk) and constructions like $\{x\}_{pk(B)}$ to represent messages. To formalize the description, we assume a set of typed variables denoted by $\mathcal{V}$. Types are implicit in our protocol descriptions, but we use the variable names as type indicators. We consider three possible types of variables: agents, nonces and keys.

For the agent type we assume a set of values (a domain) representing agent identities denoted by $\mathcal{A}$. We use variables with names like $A, B, C\ldots$ to range over agents. We reserve I for the intruder agent.

For the cryptographic primitives, we assume a set of keys $\mathcal{K}$ and use the functions $pk, sk : \mathcal{A} \to \mathcal{K}$ and $ssk : \mathcal{A} \times \mathcal{A} \to \mathcal{K}$ mapping agents to their public, secret and secret shared keys respectively. Given a key $k \in \mathcal{K}$, we denote by k^{-1} its *inverse* key defined by

$$k^{-1} := \begin{cases} sk(A) & \text{if } k = pk(A), \\ pk(A) & \text{if } k = sk(A), \\ ssk(A,B) & \text{if } k = ssk(A,B), \end{cases}$$

for some agents A and B.

A nonce is a fresh entity bound to a protocol specific execution and that cannot be reused in another one. We let the variables $Na, Nb, Nc\ldots$ range over the set of nonce values that we denote by $\mathcal{N}$.

Messages are *terms* that can be variables, or concrete values such as agents (identities), nonces, or keys. For any other type of information such as message headers, plaintext, etc., we assume a set of constants that we denote by $\mathcal{C}$. Messages can as well be constructed from other messages by encryption, grouping or function application. More precisely, we define the set of messages $\mathcal{X}$ by first adding the *ground* messages:

$$\mathcal{V} \cup \mathcal{A} \cup \mathcal{N} \cup \mathcal{K} \cup \mathcal{C} \subseteq \mathcal{X}$$

and then inductively the *constructed* messages:

$$\forall A, B \in \mathcal{V} \cup \mathcal{A} : \; pk(A), sk(A), ssk(A, B) \in \mathcal{X}$$
$$\forall x \in \mathcal{X}, k \in \mathcal{K} : \; \{x\}_k \in \mathcal{X}$$
$$\forall x_1, x_2 \in \mathcal{X} : \; (x_1, x_2) \in \mathcal{X}$$

A message is said to be *ground* if it is free from variables.

Observe that the last rule in the definition of $\mathcal{X}$ can be generalized to arbitrary tuples in a straightforward manner. In general, the way in which messages are grouped is clear and the use of parenthesis does not add any information. In such cases, parenthesis are simply omitted.

Compared to the protocol formalization in [8], our protocol definition is based on a different set of types and fixed set of functions. The message definition can be extended to other functions with arbitrary signatures provided that functions are only applied to ground messages. Despite the restrictions, many protocols (as shown in the next section) can be described using this formalism.

2.2 Protocol Examples

We consider four of the classical key distribution protocols which are two versions the Needham-Shroeder (NS) public key protocol [23], the Tatebayashi-Matsuzaki-Newman key distribution protocol [26], and a simplified version of the Denning-Sacco (DS) key distribution protocol [12].

The goal of the NS protocol is the mutual authentication of two agents: an initiator A and a responder B. The protocol relies on public key cryptography. Each agent possesses a public key. In addition, each agent shares a secret key with a trusted server S from which public keys can be retrieved. The complete version of the protocol is shown in Fig. 1 (to the left). A reduced version (to the right in the same figure) can be obtained assuming that the agents already know each others public keys.

$$\begin{array}{ll}
A \longrightarrow S : A, B & \\
S \longrightarrow A : \{pk(B), B\}_{ssk(S,A)} & \\
A \longrightarrow B : \{Na, A\}_{pk(B)} & A \longrightarrow B : \{Na, A\}_{pk(B)} \\
B \longrightarrow S : B, A & \\
S \longrightarrow B : \{pk(A), A\}_{ssk(S,B)} & \\
B \longrightarrow A : \{Na, Nb\}_{pk(A)} & B \longrightarrow A : \{Na, Nb\}_{pk(A)} \\
A \longrightarrow B : \{Nb\}_{pk(B)} & A \longrightarrow B : \{Nb\}_{pk(B)}
\end{array}$$

Figure 1: The full (left column) and reduced (right column) versions of the NS protocol

The TMN protocol is for the establishment of a secret session key between two agents A and B via a trusted server S (see Fig. 2). The protocol relies on Vernam encryption (exclusive or). First, the agents send to the server nonces Na and Nb encrypted by the server public key. Then, the server replies by the Vernam encryption of the nonces denoted by $V(Na, Nb)$ and thus each agent, knowing one of the nonces, can retrieve the other one.

$$\begin{array}{l}
A \longrightarrow S : B, \{Na\}_{pk(S)} \\
S \longrightarrow B : A \\
B \longrightarrow S : A, \{Nb\}_{pk(S)} \\
S \longrightarrow A : B, V(Na, Nb)
\end{array}$$

Figure 2: The TMN protocol

The goal of the DS protocol is the establishment of a secret key between two agents A and B. A simplified version of the protocol is shown in Fig. 3 where A sends to B a fresh secret session key $ssk(A, B)$ encrypted by A secret key and the the public key of B. Agent B replies then by a message Sec encrypted by this session key. Since $ssk(A, B)$ is secret, the message Sec should remain secret.

$$\begin{array}{l}
A \longrightarrow B : \{\{ssk(A, B)\}_{sk(A)}\}_{pk(B)} \\
B \longrightarrow A : \{Sec\}_{ssk(A,B)}
\end{array}$$

Figure 3: Simplified version of the DS protocol

2.3 Intruder Model

In the Dolev-Yao model, the attacker has full control of the communication medium. That is the attacker can intercept, drop, forward, or replay any sent message. While intercepting messages, the attacker collects knowledge. He can then use this knowledge to create and send his own messages or tamper with other agent messages.

We denote by $\mathbb{K}$ the set of all messages in possession of the attacker. During the execution of the protocol, the intruder adds to $\mathbb{K}$ any sent message. For a message x, we use $I \vdash_{\mathbb{K}} x$ to denote that the intruder can derive or learn x using his knowledge. Depending on the message type, this derivation of new knowledge is achieved using the *deconstruction* rules of Fig. 4. Using this new knowledge, the intruder can create

$$\frac{x \in \mathbb{K}}{I \vdash_{\mathbb{K}} x}\texttt{possess}$$
$$\frac{I \vdash_{\mathbb{K}} (x_1, x_2)}{I \vdash_{\mathbb{K}} x_1}\texttt{proj}_1 \qquad \frac{I \vdash_{\mathbb{K}} (x_1, x_2)}{I \vdash_{\mathbb{K}} x_2}\texttt{proj}_2$$
$$\frac{I \vdash_{\mathbb{K}} \{x\}_k \quad I \vdash_{\mathbb{K}} k^{-1}}{I \vdash_{\mathbb{K}} x}\texttt{decrypt}$$

Figure 4: Knowledge deconstruction rules

and send his own messages. Creation of new messages is based on the *construction* rules of Fig. 5.

2.4 Operational Semantics

The goal of our formal analysis is to check the security properties of secrecy and authentication. In order to formalize the properties, we need to formalize the operational semantics of security protocols. This requires defining the notion of *instantiation* by which we can create execution threads for running the different roles. First, let's fix a protocol with $n \in \mathbb{N}$ roles denoted by $P = \{r_1, r_2, \ldots, r_n\}$.

$$\frac{x \in \mathbb{K}}{I \vdash_{\mathbb{K}} x}\texttt{possess}$$

$$\frac{I \vdash_{\mathbb{K}} x_1 \quad I \vdash_{\mathbb{K}} x_2}{I \vdash_{\mathbb{K}} (x_1, x_2)}\texttt{pair} \quad \frac{I \vdash_{\mathbb{K}} x \quad I \vdash_{\mathbb{K}} k}{I \vdash_{\mathbb{K}} \{x\}_k}\texttt{encrypt}$$

Figure 5: Knowledge construction rules

For a role $r \in P$, we denote by $\mathcal{V}(r)$ the set of variables in r. As explained earlier all the variables are typed. A role $r \in P$ is instantiated by a substitution $\sigma : \mathcal{V}(r) \to \mathcal{A} \cup \mathcal{N} \cup \mathcal{K}$. We denote by $\sigma(r)$ the set of all such substitutions, i.e. $(\mathcal{A} \cup \mathcal{N} \cup \mathcal{K})^{\mathcal{V}(r)}$. We will only consider *well-typed* substitutions, i.e. mapping variables to values in their respective domains. We assume that the role action lists are indexed from 1 and let $r[i]$ denote the i^{th} action in r in case it is defined, and $\perp$ otherwise. An *instance* (or thread) of role r is a tuple (r, i, σ) where $i \in \mathbb{N}$ is the current action that can be executed (the instruction pointer) and $\sigma \in \sigma(r)$.

A transition system is a tuple $(S, s_{init}, \to)$ where S is a set of states, $s_{init} \in S$ is an initial state and $\to \in S \times S$ is a transition relation. Abusing notation, we write $s \to s'$ to denote that $(s, s') \in \to$. Given an initial set of ground messages $\mathbb{K}_{init}$ assumed to be known to the intruder, the protocol P (together with $\mathbb{K}_{init}$) induces a transition system $(S, s_{init}, \to)$ where each state in S is a tuple $(c, Th, \mathbb{K})$ such that $c \in \mathbb{N}$ is a counter used for keeping track of the number of created threads, Th is a function from $\mathbb{N}$ to role instances used to assign a unique number (identifier) to each instance, and $\mathbb{K}$, as defined earlier, is the set of messages in the possession of the intruder. For a role r and a state $s \in S$, we say that a substitution $\sigma \in \sigma(r)$ is *admissible* for s and write $\sigma \triangleright s$ iff σ does not map any of the nonce variables of r to the same value as any of the substitutions in the other instances of the same role r. Given a number $i \in \mathbb{N}$ and a role instance (r, j, σ), we denote by $Th[i \mapsto (r, j, \sigma)]$ the function defined from Th as follows:

$$\forall n \in \mathbb{N} : Th[i \mapsto (r, j, \sigma)](n) := \begin{cases} (r, j, \sigma) & \text{if } n = i, \text{ and} \\ Th(n) & \text{otherwise.} \end{cases}$$

The initial state s_{init} is defined by $(0, Th_\emptyset, \mathbb{K}_{init})$ where we use $Th_\emptyset$ to denote the role instance function that is undefined everywhere. Finally, the transition relation $\to$ is as defined in Fig. 6 by the `create`, `receive` and `send` rules.

2.5 Security Properties

In general for a sequence π, we use $|\pi|$ to denote its length (∞ if its infinite) and $\pi[i]$ to denote its i^{th} element if any, and $\perp$ otherwise. Assume a protocol P (together with a set $\mathbb{K}_{init}$) and the induced transition system $(S, s_{init}, \to)$. A *trace* τ (or run) of P is a possibly infinite sequence of states $s_1 s_2 s_3 \ldots$ such that for $s_1 = s_{init}$ and $\forall i. 1 \le i < |\tau| : s_i \to s_{i+1}$. The set of all traces of P is denoted by $Traces(P)$

We are interested in checking secrecy and authentication properties. Secrecy is related to which messages that are in the possession of the attacker. This is a state property that can be formulated as an invariant on the set of reachable states. For that let's denote by $Reach(P)$ the set of reachable states. These are the states that occur in the traces of P. Given a message x, the condition on $Reach(P)$ for the secrecy of x is given by

$$\forall s = (c, Th, \mathbb{K}) \in Reach(P) : \neg(I \vdash_{\mathbb{K}} x). \tag{2}$$

Informally, this means that the intruder is never able to learn the message x.

Authentication properties are on the other hand trace properties [7, 20]. They can be of the form "if an event happens, then another event must have happened before". We will only consider such forms. In order to be able to define such properties, we extend the role grammar with special *event* actions that are transparent to the intruder. More precisely, for a message x we use $\texttt{event}(x)$ to denote the event action with argument x. In order to handle events, we extend the transition relation $\to$ with a "non-silent" relation $\xrightarrow{x} \subseteq S \times S$ where $x \in \mathcal{X}$. This relation is defined by the `emit` rule in Fig. 6. For a trace τ of P, we use $\tau^\uparrow$ to denote the sequence of "emitted" event messages (if any) by the taken transitions. Given two messages $x, y \in \mathcal{X}$, a trace property of the form "if event x happens, then event y must have happened before" can then be formulated as follows:

$$\forall \tau \in Traces(P)\, \forall i. 1 \le i \le |\tau^\uparrow| :$$
$$\tau^\uparrow[i] = x \implies \exists j. 1 \le j \le i : \tau^\uparrow[j] = y. \tag{3}$$

We show in the example of the next section how such properties can be reduced to state properties of the form (2).

3. PROMELA MODELING

Our approach for modeling security protocols in Promela consists of three phases: the protocol phase, the intruder phase and the security properties phase. The protocol modeling phase is simple and can be generated automatically from a higher level specification language in a similar manner to how it is done in the Casper tool [19]. The two last phases are generic and the resulting models can be reused for different protocols after possibly small modifications. To illustrate the method, we use as an example the DS protocol of Fig. 3.

3.1 Protocol Model

The first step is to model the communication medium. In contrast to the method in [21], we will not consider separate channels for the different message structures. Instead, we normalize the message format and consider only one synchronous channel. By normalizing, we mean using a message length, i.e. number of fields, that can fit all possible messages in the protocol specifications. Each field in such message represents a place holder for a function application result, or a concrete value of a variable. Unused fields are filled with a padding (a constant to be defined). The enforcement of the format and the type checking of the messages is deferred to the process implementations of the roles. For a protocol P, we denote by $\sharp(P)$ this message length that we define as follows: First, for a message $x \in \mathcal{X}$, we let

$$\sharp(x) := \begin{cases} 1 \text{ if } x \in \mathcal{V} \cup \mathcal{A} \cup \mathcal{N} \cup \mathcal{C} \cup \mathcal{K}, \\ 1 \text{ if } x = pk(y) \text{ or } x = sk(y) \text{ for some } y \in \mathcal{X}, \\ 1 \text{ if } x = ssk(x_1, x_2) \text{ for some } x_1, x_2 \in \mathcal{X}, \\ \sharp(x_1) + \sharp(x_2) \text{ if } x = (x_1, x_2) \text{ for some } x_1, x_2 \in \mathcal{X}, \\ \sharp(x_1) + \sharp(x_2) \text{ if } x = \{x_1\}_{x_2} \text{ for some } x_1, x_2 \in \mathcal{X}. \end{cases}$$

Informally, $\sharp(x)$ is the number of function applications, constants, identities, nonce values, key values, and variables

$$\frac{s=(c,Th,\mathbb{K})\in S \quad r\in P \quad \sigma\in\sigma(r)\rhd s}{s\to(c+1,Th[c+1\mapsto(r,1,\sigma)],\mathbb{K})}\,\texttt{create}$$

$$\frac{s=(c,Th,\mathbb{K})\in S \quad r\in P \quad i,j\in\mathbb{N} \quad \sigma\in\sigma(r) \quad Th(i)=(r,j,\sigma) \quad x\in\mathcal{X} \quad r[j]=\texttt{recv}(x)}{s\to(c,Th[i\mapsto(r,j+1,\sigma)],\mathbb{K}\cup\sigma(x))}\,\texttt{receive}$$

$$\frac{s=(c,Th,\mathbb{K})\in S \quad r\in P \quad i,j\in\mathbb{N} \quad \sigma\in\sigma(r) \quad Th(i)=(r,j,\sigma) \quad x\in\mathcal{X} \quad r[j]=\texttt{send}(x) \quad I\vdash_{\mathbb{K}}\sigma(x)}{s\to(c,Th[i\mapsto(r,j+1,\sigma)],\mathbb{K})}\,\texttt{send}$$

$$\frac{s=(c,Th,\mathbb{K})\in S \quad r\in P \quad i,j\in\mathbb{N} \quad \sigma\in\sigma(r) \quad Th(i)=(r,j,\sigma) \quad x\in\mathcal{X} \quad r[j]=\texttt{event}(x)}{s\xrightarrow{\sigma(x)}(c,Th[i\mapsto(r,j+1,\sigma)],\mathbb{K})}\,\texttt{emit}$$

Figure 6: Transition relation

that are not used as function arguments in x. For a role $r\in P$, a message $x\in\mathcal{X}$ and an action $\texttt{act(x)}\in r$ for some $\texttt{act}\in\{\texttt{send},\texttt{recv}\}$, we define $\sharp(\texttt{act}(x)):=\sharp(x)$. We extend $\sharp$ to roles by $\sharp(r)=\max_{i\in\mathbb{N}} r[i]$ where $\sharp(\perp):=0$. Finally, we let $\sharp(P):=\max_{r\in P}\sharp(r)$. Observe that for our example, $\sharp(\text{DS})=3$. Using this definition, we can declare the communication channel for DS.

```
chan comm = [0] of {mtype, mtype, mtype};
```

The second step is to create the different variable domains. In order to do that, we define one set of names that contains all possible concrete values of the variables in the protocol assuming a fixed number of role instances. In general, given a protocol P, the corresponding set of names is defined as follows: First, we add a name representing a different agent identity for each role in P. Then we add a name for the intruder agent. If one of the roles in P contains a nonce variable, then we add names representing nonce values for each of the newly added agents (including the intruder). If one of the roles contains a key variable, then we add similarly different names representing key agents. For each different function used in P (for example *sk* or *pk*), we add names representing the values of the function application on all possible combination of arguments (for example of the added agent names). Finally, we a add a name for each different constant. The resulting set of names for the DS example is then:

```
mtype = {NULL, Sec, A, B, I, PKa, PKb, PKi, SKa
    , SKb, SKi, SSKab, SSKai, SSKbi};
```

where NULL is a special name used to fill in unused message fields (the padding constant). The names are internally represented by integer constants and the whole set mtype is represented by a range in decreasing order from the value of NULL to 1. Therefore NULL can be used to represent the name set size. In addition, macros can be used to define for example type predicates, and also to map agent names to corresponding nonces, keys, etc.

The next step is the implementation of the roles. Each role is implemented by a process taking an agent name as input. This name represents the identity of the agent running the role. The process for the initiator role takes an additional argument (agent name) representing the responder identity (intended peer). As expected, in the process of Fig. 7, there are two communication events (lines 7 and 10) corresponding to the send and receive operations in the higher level

```
1  proctype Initiator(mtype a;mtype b)
2  {
3    mtype sskab, sec;
4    atomic {
5      IniRunning(a, b);
6      SessionKey(a, b, sskab);
7      comm!kab, SecretKey(a), PublicKey(b);
8    }
9    atomic {
10     comm?sec, eval(sskab), eval(NULL);
11     IniCommit(a, b);
12   }
13 }
```

Figure 7: Initiator process for the DS protocol

specification of the protocol from Fig. 3. Observe the use of macros in lines 6 and 7, namely: SessionKey, SecretKey and PublicKey. Intuitively, the macros are the implementation of the functions from the protocol specification. Since we already created all possible names for the corresponding arguments and possible results, we only need to relate them as intended so that for example the SecretKey(A) is SKa and the SecretKey(B) is SKb, and so forth. Depending on how the names are arranged in the declaration, such macros can be implemented by simple arithmetic operations such as for SecretKey(x):

```
# define SecretKey(x)          x − 6
```

In the worst case, one has to use conditionals to map each possible combination of argument names to the intended result name such as for SessionKey(a, b, k):

```
# define SessionKey(a, b, k)
  if
  :: (a == A && b == B) || (b == B && b == A)
     −> k = SSKab
  :: (a == A && b == I) || (b == I && b == A)
     −> k = SSKai
  ...
```

In both cases, this translation process can be done automatically.

The responder process is provided in Fig. 8. As mentionned earlier, the format and type checking of the messages is deferred to the processes and hence the recurrent use of eval (lines 5 in Fig. 8 and 10 in Fig. 7), and special macros like IsSecretKey in line 6.

```
proctype Receiver(mtype b)
{
  mtype sskab, ska, pkb, sec;
  atomic {
    comm?sskab, ska, eval(PublicKey(b));
    IsSecretKey(ska);
    RecRunning(Agent(ska), b);
  }
  atomic {
    Secret(Agent(ska), sec);
    comm!sec, sskab, NULL;
    RecCommit(Agent(ska), b);
  }
}
```

Figure 8: Responder process for the DS protocol

The final step is the instantiation of the roles. This is done in the main process below. Observe that we include the possibility of the initiator starting a session with the intruder. In our models, the intruder is always given similar "credentials" (PKi, SKi, SSKai, SSKbi) as honest agents so that he can play the same roles. In such cases one can regard the intruder as impersonating compromised agents.

```
init {
  atomic
  {
    if
      :: run Initiator(A,B)
      :: run Initiator(A,I)
    fi;
    run Receiver(B);
  }
}
```

This role instantiation process is very restricted compared to the `create` rule of Fig. 6. Nevertheless, it is sufficient for finding the expected attacks (see Section 4).

3.2 Intruder Model

In the formal model of security protocols (Section 2), the intruder has an infinite memory so that he can record any exchanged messages and thus the set $\mathbb{K}$ can grow arbitrarily. Obviously, we cannot implement such model in Promela. Nevertheless, since we have a finite number of instances and we have already defined all the possible free names we can use Boolean vectors to keep track of the intruder knowledge.

```
bool Knows[NULL];
bool Keys[NULL];
```

The Keys vector is used for keeping track of the encryption keys in the possession of the intruder (like public ones). The Knows vector is used in a similar manner for all other types of names but for keys it has different implications; more precisely it means that the intruder knows the corresponding inverse keys. Observe that one can exploit the fact that the set of names is a range and use the names directly as indices for access.

The vectors are intended to be updated upon each interception of a message. This does not provide enough support for fully implementing the intruder capability defined by the `receive` rule of Fig. 6. In fact the other issue that needs to be solved is the capacity of the intruder to store messages "unknown" to him such as messages encrypted with keys he doesn't possess. In the formal model, the intruder can store such messages so that he can later forward or use them in his own created messages (replay attacks). In our case, we will consider a finite memory intruder with the capacity of storing at most one message.

The intruder is implemented in a separate process shown in Fig. 9. The main loop contains three statements. The first

```
1  proctype Intruder()
2  {
3    mtype d = NULL, k1 = NULL, k2 = NULL;
4    mtype pd = NULL, pk1 = NULL, pk2 = NULL;
5    do
6      :: comm?d, k1, k2 ->
7         atomic {
8           AddToKnowledge(d, k1, k2);
9           if
10            :: skip
11            :: pd = d; pk1 = k1; pk2 = k2
12          fi;
13          if
14            :: skip
15            :: comm!d, k1, k2
16          fi
17        }
18     :: RandMessage(d, k1, k2) ->
19        atomic {
20          IsValidMessage(d, k1, k2, pd, pk1, pk2
              ) -> comm!d, k1, k2
21        }
22     :: comm!pd1, pd2, pk
23   od
24 }
```

Figure 9: The intruder process

one (lines 6-17) implements the message interception capability in the following steps: First (line 8), the intruder updates his knowledge using a function called AddToKnowledge which simply updates the Keys and Knows vectors. Second, he can choose to either update his memory by storing the message thus erasing any previously stored one (line 10) or not (11). Last, the intruder can choose to drop the intercepted message (14) or forward it (15).

The second statement (lines 18-21) implements the message creation and injection capability. For that, the intruder uses a function called RandMessage for message creation and a macro called IsValidMessage for checking: a) the format and the field types of the created message, and b) its consistency with the intruder knowledge. In particular, this last check is needed in order to avoid false attacks and thus is important for completness.

The third statement (line 22) simply sends the stored message hence partly implementing the intruder capability of replaying old messages.

The intruder process is generic and can be reused as is modulo small modifications for adapting it to protocols with different message length. The choice of the local variable names, which can be arbitrary, was based on the expected field types only for clarity. The function for the random message creation can be also generic as illustrated in Fig. 10 but we have also experimented with other variants (Section 4).

```
inline RandMessage(d, k1, k2)
{
  atomic {
    do
      :: ( d < NULL ) -> d = d + 1
      :: ( d > 1 ) -> d = d - 1
      :: ( k1 < NULL ) -> k1 = k1 + 1
      :: ( k1 > 1 ) -> k1 = k1 - 1
      :: ( k2 < NULL ) -> k2 = k2 + 1
      :: ( k2 > 1 ) -> k2 = k2 - 1
      :: break
    od;
  }
}
```

Figure 10: Generic random message creation function

The parts that are more related to the protocol specifications are shown below. The AddToKnowledge function implements the message deconstruction rules of Fig. 4.

```
inline AddToKnowledge(d, k1, k2)
{
  if
    :: IsKey(k2) && Knows[k2 - 1] &&
       IsSecretKey(k1) ->
       Knows[d - 1];
       if
         :: IsKey(d) -> Keys[d - 1] = 1
         :: else skip
       fi
    :: IsNULL(k2) && IsKey(k1) && Knows[k1 - 1]
       ->
       Knows[d - 1] = 1
    :: else skip
  fi;
}
```

While the IsValidMessage macro checks the construction rules of Fig. 5. In this particular case, we allow the construction of messages using information from the stored one which is passed to the macro in the additional arguments pd, pk1, pk2.

```
# define IsValidMessage(d,k1,k2,pd,pk1,pk2)
    ((IsKey(k2) && Keys[k2 - 1] &&
      IsSecretKey(k1) || IsNULL(k2)) &&
     Keys[k1 - 1] && Knows[d - 1] ) ||
    (IsKey(k2) && Keys[k2 - 1] &&
     k1 == pk1 && d == pd)
```

Observe that all of the intruder related functions and macros can also be automatically generated from any higher level specification of protocols that is based on the notations and definitions we present in Section 2.

Finally, now that we have defined the intruder model, what remains to do is to update the main process in order to first define the intruder initial knowledge and then run the corresponding process. The definition of the initial knowledge is achieved by setting the known names'values to *true* in the knowledge vectors (Knows and Keys). In the DS case, all the names, except Sec, SSKab, SKa, SKb, are initially known to the intruder.

3.3 Security Properties

We use active processes to check for the security properties which we implement using macros and some additional global variables.

A secrecy property can be checked by querying the knowledge of the intruder (see (2) in Section 2). In our modeling approach, we capture this knowledge in the Boolean vector Knows. Therefore, for the DS protocol example, an invariant for the secrecy of the Sec message can be defined as follows:

```
# define SecInv (!Knows[Sec - 1])
```

A possible implementation of a process for checking this invariant is given below.

```
active proctype SecMonitor()
{
  atomic {
    do
      :: !SecInv -> assert(SecInv)
    od
  }
}
```

In order to check authentication, we reduce trace properties of the form of (3) to state properties in the Promela models. We do that following the methodology of [21] which makes use of global variables to record the occurrence of the events that are of interest. The property is then expressed as a Boolean condition on those variables. Such condition can be monitored in a similar manner to how it is done for secrecy.

```
active proctype AuthMonitor()
{
  atomic {
    do
      :: !AuthInv -> assert(AuthInv)
    od
  }
}
```

In general, we will assume that protocols, such as the ones we consider always contain an initiator role and a responder role that need to authenticate each others. We consider two events: *Running* and *Commit* indicating that an agent has respectively initiated and completed a protocol run with another agent. This technique is well described in Lowe's works [20, 18]. We then introduce four global Boolean variables for authentication in order to record the corresponding Running and Commit (implicit) events in the model.

```
bit IniRunningAB = 0;
bit IniCommitAB = 0;
bit ResRunningAB = 0;
bit ResCommitAB = 0;
```

We use then different macros to update the authentication variables at specific points during the execution of the protocol.

```
# define IniRunning(x, y)
  if
  :: ((x == A) && (y == B)) -> IniRunningAB = 1
  :: else skip
  fi
# define IniCommit(x, y)
...
```

Finally, the authentication invariant can be defined as follows:

```
# define AuthInv
  ( (!IniCommitAB || RecRunningAB) &&
    (!RecCommitAB || IniRunningAB) )
```

Observe that the only parts that are specific to the protocol in the implementation of the security properties are the definition of the secrecy invariant and the placement of the authentication macro calls in the role processes.

4. EXPERIMENTAL RESULTS

Using our methodology, we have implemented all the protocol examples of Section 2. We have also experimented with different implementation variants of the function for the random message creation. SPIN was able to find all the expected attacks. We used a laptop with an i5 Intel processor where we executed SPIN in a 1GB RAM virtual machine running a 32 bit version of Linux. All the results are summarized in Table 1. We run SPIN with the bitstate storage and breadth-first search modes. After each unsuccessful run of SPIN, we keep doubling the estimated state space until either an attack is found or the memory limit is reached indicated respectively by "a" or "u" in the table. Some of the found attacks are show in figures below (See Fig. 11, Fig. 12, Fig. 13 and Fig. 14).

In Table 1, the column headers denote the different used implementation variants of the message creation function RandMessage. The variant ATT_1 corresponds to the one shown in Fig. 10. ATT_2 is an alternative version which exploits the intruder current knowledge in order to restrict the number of available choices in the loop. In this version, we use Boolean variables to prevent recurrent modifications of the same field in contrast to ATT_1 where any field can be modified several times. Each of the additional variables is used as guard which is set as soon as the corresponding field is modified preventing any further changes. Notice that since in ATT_2, we are exploiting the intruder knowledge, we need to allow the creation of messages containing unknown data from the stored one. Therefore, the function requires additional arguments for passing the stored message data. We give below an extract of the implementation of ATT_2 for the DS protocol example.

```
inline RandMessage(d, k1, k2, pd, pk1, pk2)
{
  atomic {
    bool dSet = 0, k1Set = 0, k2Set = 0;
    do
      :: !dSet -> d = NULL; dSet = 1
      :: Knows[A - 1] && !dSet -> d = A; dSet =
           1
      :: Knows[B - 1] && !dSet -> d = B; dSet =
           1
      ...
```

The ATT_3 variant is shown below. The motivation behind it was to make a generic version of ATT_2 that is independent of the protocol implementation names. This is can be done by further exploiting the fact that the name set is a range. The idea is to alter ATT_1 so that unknown names are skipped. In the version below we skip one unknown name at a time (lines 8-9). But this can be further developed in order to skip arbitrary number of adjacent unknown names in the range. Such code would be complex but still can be automatically generated.

```
1  inline RandMessage(d, k1, k2, pd, pk1, pk2)
2  {
3    atomic {
4      do
5        :: (d < NULL && (Knows[d] || d + 1 == pd)
             ) -> d = d + 1
6        :: (d > 1 && (Knows[d - 2] || d - 1 == pd
             ) -> d = d - 1
7        ...
8        :: (d < NULL - 1 && !(Knows[d] || d + 1
             == pd) ) -> d = d + 2
9        :: (d > 2 && !(Knows[d - 2] || d -1 == pd
             ) ) -> d = d - 2
10       ...
```

The final variant we consider is ATT_4. It is derived from the previous one by simply eliminating the statements for skipping unknown names (lines like 8-9). This is an unsound modification because the function can no longer scan the whole range and the choices for the new names are restricted to only the adjacent known ones. Surprisingly, this is the only variant which was efficient enough so that the attack on the full NS protocol could be found as well.

Table 1: Experimental results: (a) for attack and (u) for unresolved

Protocol	ATT_1	ATT_2	ATT_3	ATT_4
DS	a	a	a	a
Reduced NS	a	a	a	a
Full NS	u	u	u	a
TMN	a	a	a	a

5. CONCLUSION

We have considered a restricted class of security protocols and gave a precise formalization of the corresponding operational semantics using transition systems. This is to motivate the use of state based model checkers such as SPIN. We have then described a method for implementing and analyzing such class of security protocols in SPIN. The description was informal, but as future work the method can be formalized and implemented in an automatic translator tool. The intruder model implementations we provide are simple, generic and sufficiently efficient so that SPIN could find all the expected attacks on several of the classical key distribution protocols. One possible direct extension of this work could be to consider how to adapt the approach to other types of message variables such as tickets, time-stamps or counters. Another one could be to analyze to which extent the method can scale to arbitrary (up to a fixed bound) number of agents each running one of the protocol roles.

6. REFERENCES

[1] 3GPP The Mobile Broadband Standard. `http://www.3gpp.org/specifications/`.

[2] *10th Computer Security Foundations Workshop (CSFW '97), June 10-12, 1997, Rockport, Massachusetts, USA.* IEEE Computer Society, 1997.

[3] *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 11-13 June 2001, Cape Breton,*

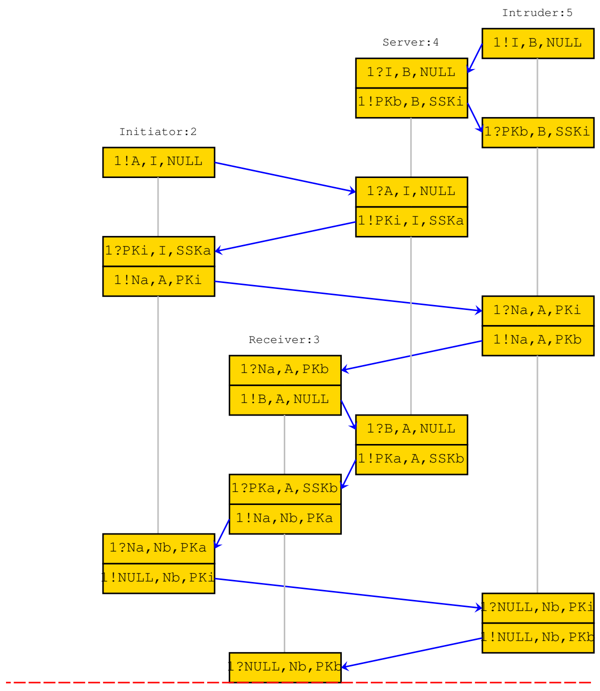

Figure 11: A trace for an attack on the full NS

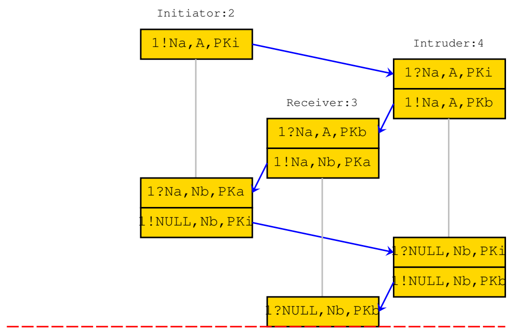

Figure 12: A trace for an attack on the reduced NS

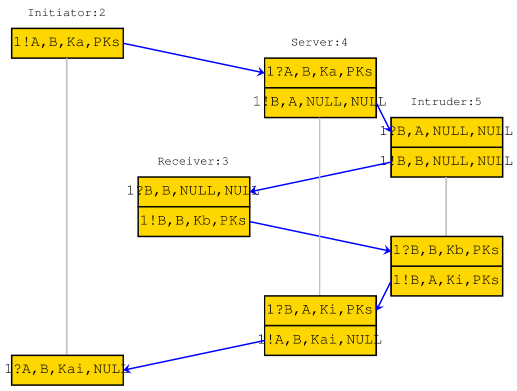

Figure 13: A trace for an attack on TMN

Nova Scotia, Canada. IEEE Computer Society, 2001.

[4] M. Abadi, B. Blanchet, and C. Fournet. Just fast keying in the pi calculus. In D. A. Schmidt, editor, *ESOP*, volume 2986 of *Lecture Notes in Computer Science*, pages 340–354. Springer, 2004.

[5] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In C. Hankin and D. Schmidt, editors, *POPL*, pages 104–115. ACM, 2001.

[6] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *CSFW* [3], pages 82–96.

[7] B. Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, 2009.

[8] C. Cremers and S. Mauw. *Operational Semantics and Verification of Security Protocols*. Information Security and Cryptography. Springer, 2012.

[9] C. J. F. Cremers. The scyther tool: Verification, falsification, and analysis of security protocols. In A. Gupta and S. Malik, editors, *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 414–418. Springer, 2008.

[10] C. J. F. Cremers. Session-state reveal is stronger than ephemeral key reveal: Attacking the naxos authenticated key exchange protocol. In M. Abdalla, D. Pointcheval, P.-A. Fouque, and D. Vergnaud, editors, *ACNS*, volume 5536 of *Lecture Notes in Computer Science*, pages 20–33, 2009.

[11] C. J. F. Cremers. Key exchange in ipsec revisited: Formal analysis of ikev1 and ikev2. In V. Atluri and C. Díaz, editors, *ESORICS*, volume 6879 of *Lecture Notes in Computer Science*, pages 315–334. Springer, 2011.

[12] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Commun. ACM*, 24(8):533–536, 1981.

[13] D. Dolev and A. C.-C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.

[14] N. A. Durgin, P. Lincoln, and J. C. Mitchell. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12(2):247–311, 2004.

[15] G. J. Holzmann. The model checker spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[16] M. Jakobsson and S. Wetzel. Security weaknesses in bluetooth. In *Proceedings of the 2001 Conference on Topics in Cryptology: The Cryptographer's Track at RSA*, CT-RSA 2001, pages 176–191, London, UK, UK, 2001. Springer-Verlag.

[17] A. S. Khan, M. Mukund, and S. P. Suresh. Generic verification of security protocols. In P. Godefroid, editor, *SPIN*, volume 3639 of *Lecture Notes in Computer Science*, pages 221–235. Springer, 2005.

[18] G. Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In T. Margaria and B. Steffen, editors, *TACAS*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer,

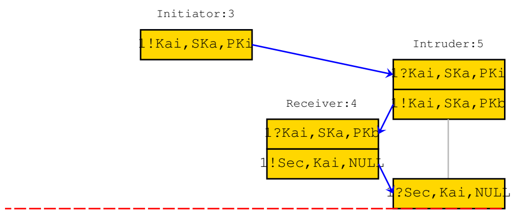

Figure 14: A trace for an attack on DS

1996.

[19] G. Lowe. Casper: A compiler for the analysis of security protocols. In *CSFW* [2], pages 18–30.

[20] G. Lowe. A hierarchy of authentication specification. In *CSFW* [2], pages 31–44.

[21] P. Maggi and R. Sisto. Using spin to verify security properties of cryptographic protocols. In D. Bosnacki and S. Leue, editors, *SPIN*, volume 2318 of *Lecture Notes in Computer Science*, pages 187–204. Springer, 2002.

[22] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin. The tamarin prover for the symbolic analysis of security protocols. In N. Sharygina and H. Veith, editors, *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 696–701. Springer, 2013.

[23] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.

[24] M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is np-complete. In *CSFW* [3], pages 174–.

[25] C. Tang, D. A. Naumann, and S. Wetzel. Symbolic analysis for security of roaming protocols in mobile networks - [extended abstract]. In M. Rajarajan, F. Piper, H. Wang, and G. Kesidis, editors, *SecureComm*, volume 96 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 480–490. Springer, 2011.

[26] M. Tatebayashi, N. Matsuzaki, and D. B. N. Jr. Key distribution protocol for digital mobile communication systems. In G. Brassard, editor, *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 324–334. Springer, 1989.