VERSITA

Annales UMCS Informatica AI XI, 3 (2011) 27–40
DOI: 10.2478/v10065-011-0002-y

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI
http://www.annales.umcs.lublin.pl/

Effective reduction of cryptographic protocols specification for model-checking with Spin

Urszula Krawczyk[1*], Piotr Sapiecha[1,2]

[1] *Krypton-Polska, Al. Jerozolimskie 131 Warsaw, Poland*
[2] *Department of Electronics and Information Technology, Warsaw University of Technology, Warsaw, Poland*

Abstract

In this article a practical application of the *Spin* model checker for verifying cryptographic protocols was shown. An efficient framework for specifying a minimized protocol model while retaining its functionality was described. Requirements for such a model were discussed, such as powerful adversary, multiple protocol runs and a way of specifying validated properties as formulas in temporal logic.

1. Introduction

A flaw in a cryptographic protocol may become a real security thread [**1, 2**]. Even a seemingly small protocol may produce a great number of possible behaviours. One of the methods to formallycon sider protocols correctness is *model checking* by representing the protocol as *Büchi* automata M, specifying every checked property as an *LTL* temporal formula α and checking satisfiability of the formula in the model $M \models \alpha$ [**3, 4, 5, 6**].

The automata of the protocol is typically generated from a more high-level description. This article has its focus on representing protocol models in the

*E-mail address: U.Krawczyk@krypton-polska.com

Promela language, which is an input for model checker *Spin* [**7**]. Choice of that tool was due to its effective, automatic minimizing techniques and its widespread use in the area of verification (see annual workshops page [**8**]).

Some examples of verifying cryptographic protocols with *Spin* can be found in the literature [**9, 10, 11, 12, 13, 14, 15**]. However, the presented models are too simple, not taking into consideration multiple runs or limiting the protocol attacker abilities. Most importantly they create a large automaton, even though not so complicated *Needham–Schroeder* protocol is considered. Such an approach for modelling a more complex protocol like *JFK* would result in a model not feasible to verify. Model publicized in [**10**] seems to be the most sophisticated, as it is scalable and includes parallel runs but it contains many redundant transition causing state–space explosion. Another approach presented in [**15**] uses interesting recursive structures but at the expense of great memory cost. This does not disable the possibility of finding the attacks but only full coverage of reachable states can assure the model behaviour correctness.

Also none of the mentioned models supports creating a readable counterexample indicating an attack. In this article a method for developing cryptographic protocol models, avoiding those drawbacks is outlined. The description of protocol framework is illustrated with the fragments of *Promela* code.

2. Problem Definition

Key establishment and authentification cryptographic protocols, such as *Needham–Schroeder* or *JFK*, can be modelled as automata so that their properties, described as temporal formulas, can be checked. The main problem is to keep such a model effectively verifiable. The satisfiability of the formulas in the model should increase confidence in the security of the protocols. Thus it is crucial to explicitly list requirements such a model must comply with. The environment in which the protocol is studied is considered an important matter [**16, 17, 18**]. The main points are the following:

- **Legal users** - can participate in parallel protocols taking different roles (initiator, responder). They can establish a session with other users including the intruder, that has a certificate like other legitimate users.
- **The intruder** - can at any point eavesdrop a message, alter it and resend it to another user in another protocol run. The adversary produces messages on the basis of his actual knowledge, creating new complex elements (e.g. encryptions) or resending the remembered ones.
- **Model scalability** - concerns the number of protocol runs and the attackers knowledge database.

: Additional data - the information required for logical assertions that are written to check protocol properties must be stored. Also additional informations about protocol state are printed out and used later while producing a counterexample.

: Model configuration - description of a particular model configuration, should specify any constraints in the way that the messages are sent from the user to the user and the roles the users can take.

These specifications are responsible for models proper behaviour. While the above constraints hold, one important parameter must be minimized:

: Models size - affects the amount of memory and time needed for verification. Considering the exponential complexity of the model checking problem ($\mathcal{O}(\#(M)) = \mathcal{O}(2^{\#(\mathcal{P})})$, where $\mathcal{P}$ is the number of atomic prepositions describing the states of model M [**19**]), this seems to be a critical issue in practical applications.

3. Representing Protocol as an Automaton

To illustrate the idea of modelling protocols as *Büchi* automata, an example is given in this section, showing a path from a protocol description up to the automaton. A clear, simple protocol is used (Fig. 1). Also no reduction techniques have been demonstrated yet. This keeps the model comprehensible so that the reader can understand the general methodology. The verification process consists of the following steps.

(1) **Modelling protocol** - the verifier describes in the *Promela* language the behaviours of the protocol users and all the possible actions the adversary can take. A sample code representing the responder in the example protocol is shown in Fig. 1.

(2) **Protocol as automaton** - the *Promela* code describes an automaton. A gard and an action are associated with every transition from state to state. In the automaton in Table 1 and Fig. 2, the state when the key is established can be reached only if the guard corresponding to signature correctness holds. The actions can change the variables values and message channels contents.

(3) ***Kripke* structure** - incorporating variables values into automaton produces a *Kripke* structure [**4, 6**]. Here every state represents a possible configuration of variables values. The example structure is shown in Fig. 3.

(4) ***Büchi* automaton** - nondeterminism of *Büchi* automaton is crucial for model checking, as every possible path in the protocol must be analyzed. *Büchi* automaton can be constructed from *Kripke* structure

by copying the state labels onto the outgoing arcs [**6**], which can be seen in Fig. 4.

(5) **The verified property** - all the desirable properties of the protocol are written down as *LTL* logic formulas. The formulas contain references to variables from the protocol model. Each formula is negated to denote the unsafe states and automatically transformed into special *never* process in the *Promela* code with *Spin* or another tool [**20**], as shown in Fig. 5. This code can be also transformed into the *Büchi* automaton. Locations represented as double framed circles are accepting locations. The automaton accepts an infinite input if it makes the automaton visits accepting states infinitely often [**5, 6**].

(6) **Verification algorithm** - at the end an asynchronous product of all automata representing protocol users is constructed. This automaton is used to construct a synchronous product with the formula automaton [**6**]. The algorithm is to search the resulting automaton for a path that would traverse infinitely often through the formula automaton accepting locations [**4, 6**].

(7) **Counterexample** - such a path indicates an error in the protocol and presents a way an unsafe state can be reached. On the whole, the protocol is flawed if its model can produce a path, that is accepted by the automaton representing an undesirable situation.

The human verifier takes part only in the stages involving modelling the protocol in the *Promela* language and writing *LTL* logic formulas. Other activities are done automatically by the model checker tool. Actually effective implementations merge the described stages to reduce computing costs.

Table 1. Table with automaton describing responders states while participating in the protocol from Fig. 1.

Transition	Current state	Gard	Transition effect	Next state
t43	0	-	-	1
t44	1	-	m1?certi,expi	2
t45	2	-	printf('MSC: MSG2 Bob...')	3
t46	3	-	m2!B, ExpB, BPr, expi, ExpB	4
t47	4	-	m3?sigkey,sigexpi,sigexpr	7
t48	7	(((certi == A ...)	-	6
t49	7	(((certi != A ...)	-	13
t1	6	-	skip;	11
t50	11	-	d_step{...}	12
t51	12	-	-	13

Modelling protocol

$$\begin{aligned}
&\text{MSG1 } A \to B: && A,\ k_a\\
&\text{MSG2 } B \to A: && B,\ k_b,\ SIG\{BPr\}(k_a, k_b)\\
&\text{MSG3 } A \to B: && SIG\{APr\}(k_a, k_b)
\end{aligned}$$

$$\begin{aligned}
&k_a = g^a \ mod\ p\\
&k_b = g^b \ mod\ p\\
&Key = k_a^b \ mod\ p = k_b^a \ mod\ p
\end{aligned}$$

```
/* responder */
proctype Bob(chan m1, m2, m3)
{
        byte expi; /* D-H exponent of initiator */
        byte certi;/* certificate of initiator */
        byte sigkey, sigexpi, sigexpr; /* for values from sign */
MSG1:   m1?certi, expi;
MSG2:   printf("MSC: MSG2 Bob %d, %d, Sig %d(%d, %d)\n",
                        B, ExpB,  BPr, expi, ExpB);
        m2!B, ExpB,  BPr, expi, ExpB;

MSG3:   m3?sigkey, sigexpi, sigexpr; /* siga */
        if  /* check sign */
            ::(( (certi == A && sigkey == APr)
                 || (certi == E && sigkey == EPr) )
                && sigexpi == expi  &&  sigexpr == ExpB )
                    -> skip;
        fi;
FINISH: d_step{ /* remember the data for verification in global variables */
            explb = expi;
            certInit = certi;
        }
}
```

Fig. 1. Description of examplary key establishment cryptographic protocol based on the *Diffie–Hellman* schema and signature and the *Promela* code representing the responder.

4. Our approach

The most intuitive way to model protocol is to represent users as independent processes, sending messages through channels controlled by the intruder.

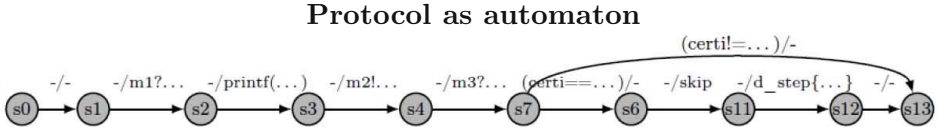


Fig. 2. Graphical representation of automaton from Table 1, describing responders' behaviour.

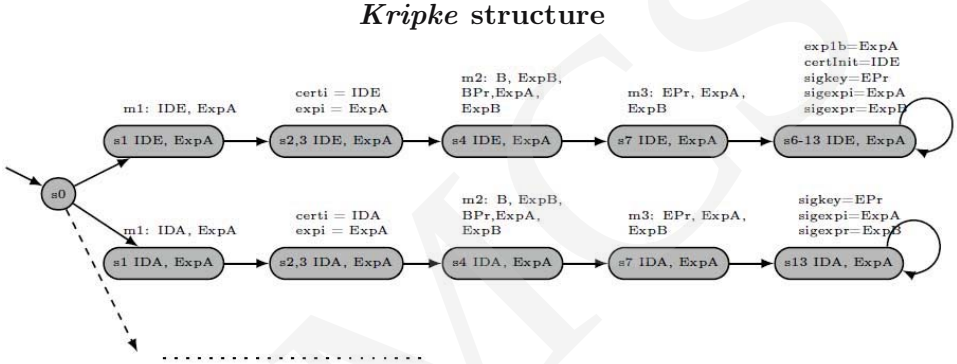


Fig. 3. *Kripke* structure constructed from automaton from Fig. 2.

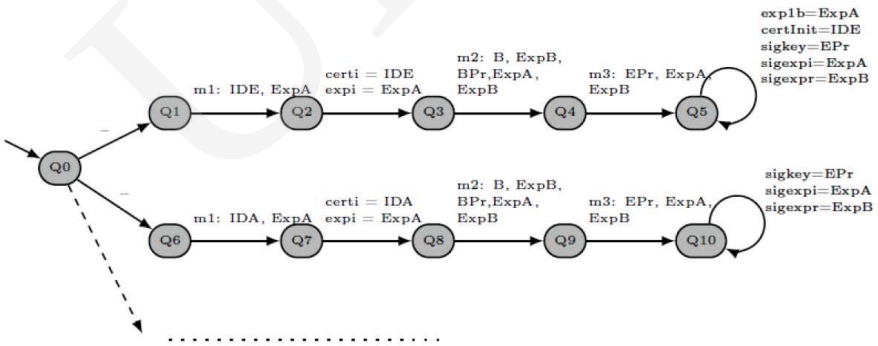


Fig. 4. *Büchi* automaton constructed from the *Kripke* structure from Fig. 3.

Unfortunately, such a model, though properly describing the protocol, might be too large to analyze. Due to exponential complexity of the problem [**19**], every redundancy in the model is expensive by means of memory and computation time.

So the ability to model a protocol is not sufficient for practical verification. Thus the constructions below were used in the presented model to reduce its complexity, while giving the intruder strong abilities.

The verified property

```
#define successAB (exp2a == ExpB  && exp1b == ExpA)
#define id_misbind  (certInit == E && successAB)
never {
T0_init:
    if
    :: ((id_misbind)) -> goto accept_all
    :: (1) -> goto T0_init
    fi;
accept_all:
    skip
}
```

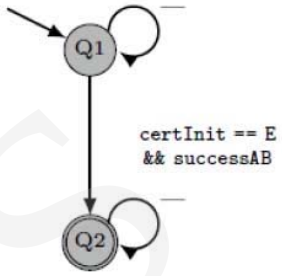


Fig. 5. *Büchi* automaton constructed from the *LTL* logic formula $\Diamond$ *id_misb* (identity misbinding attack is possible). Specification in the *Promela* code (left) and graph representation (right).

4.1. Remembering Simple Message Elements

Simple elements known by the intruder are remembered as bytes in the `EveDB` array. Every element has unique value and can be accessed with a combination of defined indices. The values for the *JFKi* protocol are shown in the left column of Fig. 6. The example of access to elements can be found in the right column of Fig. 6. For instance the index of responders nonce `nonr`, is a sum of index indicating user identifier, *nonce* type and current protocol run ($otherUser + NONCE + comm$). On the other hand, exponentials are reused between protocol runs so to access them the `comm` variable indicating the run is not used. If the `EveDB` array cell is not empty, the value is known by the attacker.

4.2. Remembering Complex Message Elements

Complex elements such as signatures and encryptions are stored by the intruder in additional channels which work like *FIFO* queues. While generating a faked message, needed elements are randomly chosen from channels. The examplary usage was shown in Fig. 6 (right). Channel `EveSig2` holds signs from the second protocol message, that were intercepted earlier.

4.3. Eavesdrop On Send, Corrupt on Receive Tactic

In a simple model the message is produced by the legal user, the intruder learns it and then the message is sent. Yet before the receiver gets it, the data is intercepted and generated once more by the attacker on the basis of their knowledge. An observation can be made, that it makes no sense to transport via the channels the data that is already stored in the intruder database. It can be seen that the original message is not used after the intruder learns it.

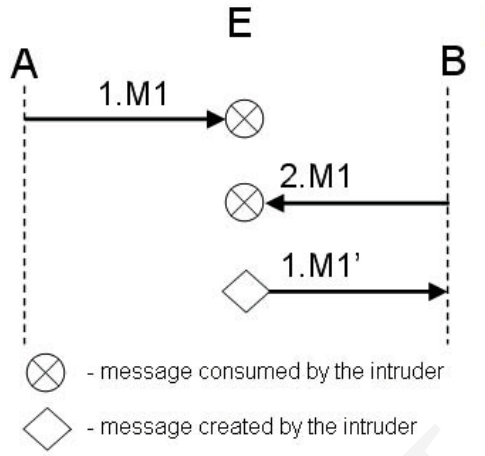
```
#define NONE 0          /* indicating element is not known */
/* Alice elements */
#define IDA 1           /* Alice certificate */
#define PrA 2           /* Alice private key */
#define ExpA 3          /* Alice exponenctial */
#define NA1 4           /* nonce used in the first protocol run */
#define NA2 5
#define HMACA 6         /* secret key for computing mac Cookie */
#define SaA1 7          /* security association in first run */
#define SaA2 8
/* Bob elements ... */
#define GrInf1 17       /* D-H group info for first run */
#define GrInf2 18       /* and in the second run */

#define IDE 19          /* intruder certificate */
#define PrE 20          /* intruder private key */
#define ExpE 21         /* intruder exponenctial */
#define NE 22           /* intruder nonce used in both runs */
#define HMACE 23        /* intruder key for computing Cookie */
#define GrInfE 24       /* D-H group faked by the intruder  */
#define SaiE 25         /* faked initiator security association */
#define SarE 26         /* faked responder security association */
/* Indexes to access elements of chosen  user*/
#define NONA    4   /* index of Alices nonce */
#define NONB    12  /* index of Bobs nonce */
#define SAA     7 /* index of security association of Alice */
#define SAB     15 /* index of security association of Bob */

#define PRKEY   1 /* offset of private key after certificate */
#define EXPON   2 /* offset of exponential after certificate */
#define NONCE   3 /* offset of nonce after certificate */
#define HMACKEY 5 /* offset of key for MAC after certificate */
#define SECA    6 /* offset of security association after cert */
#define GRP_INF 17 /* offset of DH group */
/* defines describing communication number */
#define COMM1   0
#define COMM2   1
```

```
if      /* intruder chooses responders nonce */
    ::nonr = intruder + NONCE;  /* intruders own nonce */
    /* nonce of the  other user that is known */
    ::(EveDB[(otherUser + NONCE + comm)] != NONE)
        -> nonr = (otherUser + NONCE + comm);
fi;
if      /* intruder chooses responders exponential */
    ::expr = intruder + EXPON;
    ::(EveDB[otherUser + EXPON] != NONE)
        -> expr = (otherUser + EXPON);
fi;
if  /* ----- SIGN1 ----- */
    ::/* compute my own sign1 */
        cert = intruder;
        prkey = intruder + PRKEY;   /* private key of the user
        /* take the chosen expr for consistency */
        sigexpr = expr;
        /* D-H group information was chosen same way as expr */
        siggrinf = grinf;
    ::(len(EveSig1) > 0) -> /* resend some intercepted sign1 */
        if
            ::(EveDB[otherUser] != NONE)
                -> cert = otherUser;
        fi;
        pom = 0;
        do
            ::(pom < len(EveSig1)) ->
        d_step{
                EveSig1? prkey, sigexpr, siggrinf;
                EveSig1! prkey, sigexpr, siggrinf;
                pom = pom + 1;
        }/* d_step*/
            ::(pom > 0) -> /* get at least one */
                break;
        od;
fi;
/* choosing a cookie from those intercepted earlier ... */
printf("MSC: MSG2.%d   Eve(%d) -> %d    "
    "%d, %d, %d, %d, %d, %d,  Sig %d(%d, %d),"
    " cookie HMACK %d (%d, %d, %d, %d)\n",
    comm, usr1, usr2,   expi, expr, noni, nonr,
    grinf, cert,  /* responder cert */
        prkey, sigexpr, siggrinf,
        hmac, hexpi, hexpr, hnoni, hnonr);
```

Fig. 6. Representation of simple message elements (left) and intruder preparing faked MSG2 in *JFKi* (right) in *Promela* language code.

Therefore in our approach channels transport only information that a message is sent as shown in Fig. 7. In consequence, all channels memory usage size is constant and small. Thus the tactic is crucial for minimizing the model size.

It is also important that in our model the intruder can produce faked message after arbitrary time, possibly after receiving other messages from parallel protocol runs and learning new data. This models the ability of the intruder to delay messages. In Fig. 7, a circle is a point where a message is consumed by the attacker, while a square marks creation of a message by him. As can be seen message `M1'` is produced after learning message `M2` from the second protocol run. Sending of message is also the point where the intruder decides where the message will be sent.




```
inline recvMSG1sendMSG2(
    self, otherLegalUsr, ...){
        /* intruder creates possibly faked MSG1'... */
        /* self deal with MSG1'... */
        /* self creates MSG2... */
        /* intruder learns MSG2... */
        /* intruder redirects MSG2: */
        /* choose protocol run */
        CHOOSE2(comNum, COMM1, COMM2)
        /* choose receiver */
        CHOOSE2(usr1, self, otherLegalUsr)
        printf("redirecting MSG2 to %d "
            "run %d\n", usr1, comNum);
        /* MSG2 sent - very small channel */
        m2!comm, usr1;
    }
    /* How recvMSG1sendMSG2() is used: */
    ::m1?[comNum, IDA] -> /* Alice tries to receive */
        m1?? comNum, IDA; // self
        recvMSG1sendMSG2(IDA, IDB, ...)
```

Fig. 7. Eavesdrop on send, corrupt on receive schema and specification in the *Promela* language.

The effect is achieved by combining attackers' activities with the users' steps, rather than putting them into a separate process. As the method name suggests the intruder takes his first action (eavesdropping) just after the legal user sends a message. The instructions are put into the sender process. The attackers' second action (corrupting the message) is put into receivers' process, just before the legal user gets a message.

The tactic also eliminates introduction of additional mechanism to prevent the intruder from intercepting his own, faked messages. This could have been an additional field in a message, indicating if the message was sent by the attacker that can be found in the literature [**10**]. With the tactic this is not needed, as the data is generated only once before the legal user receives it.

4.4. Only One Channel For a Message

Using for every message two channels (first for transporting data from the legal user to the intruder, second for transporting data to the legal receiver) is simple and intuitive but memory expensive. Thus only one channel is used in our model. This can be done as no message data is really transported as was mentioned. Only information that a message is sent is placed in channels.

4.5. All Users in One Process

The *eavesdrop on send, corrupt on receive* technique also makes it possible to place the code of all users in one process. As it was mentioned, the intruders' actions are combined with those of legal users. In a simple model senders and receivers could be put in independent processes. Every step of a user consists

of receiving and/or sending a message. Such step would be put into an `atomic` clause to minimize interleavings we are not interested in. This would result in a sequence of users' atomic steps from different protocol runs, which is the models proper behaviour.

Yet the presence of many processes would cause the model checker to create an asynchronous product of the automata. This would introduce redundant interleaving and make the verified model grow too much [**3**]. That is why only one process is used with a `do` loop, in which from the set of executable steps one is nondeterministically chosen. Every step is represented by a function to keep both the advantage of one process and of having structured the *Promela* code, as shown in Fig. 7. Rather than storing the user identity in his process state, it becomes the function parameter. For example, a receiver of the message is indicated by the `self` parameter of function `recvMSG1sendMSG2()`.

In such a model a sequence of nondeterministically chosen steps is produced just as in the multi-process case but without the undesirable overhead. So this approach does not affect the models functionality but its efficiency.

4.6. Consistent Message Generation by the Intruder

The consistent generation of messages means that once chosen, an element (e.g. *nonce*, exponential) is used by the intruder in the whole message. This helps avoid messages that are known to be rejected by legal users. The example of this was shown in Fig. 6 (right). Here the same value is used as exponential of a responder `expr` in the plain text and in the faked signature.

5. Protocol Properties Verification

The last step is specifying protocol properties as *LTL* (linear temporal logic) formulas. Notation $\Box\alpha$ means that α is satisfiable in the model, iff it is true for every execution path of the automata. It can be used to specify that it is desirable that unsafe states are never reachable. For the *Needham–Schroeder* protocol, an example safety formula detecting identity misbiding would be:

$$\begin{aligned}\alpha = \big(& run2accepted \;\&\& \\ & (otherUsrA[COMM1] == IDB \,\&\&\, otherUsrB[COMM1] == IDE \\ & \| \; otherUsrA[COMM1] == IDE \,\&\&\, otherUsrB[COMM1] == IDA) \big)\end{aligned}$$

The wrong state is when one of the legitimate users accepts a session with another legal user (IDA or IDB), while this user did not, because he was engaged in a run with the intruder (IDE). So it should hold that $M \models \Box\neg\alpha$. Fig. 8 presents a readable counterexample for the attack. It was automatically produced by a simple driver written by the authors, that runs the model checker, parses *Spins* output and interprets it. Only the emphasis was added by hand

for more readability. The required information, from the raw output of the model checker, originates from the printing commands shown in Figs 1 and 6. Form the listing it can be seen that *Bob* accepted a session with *Alice* who never took part in a run with *Bob*.

```
2  m1!0,1
   MSC: MSG1.1   1 -\9 Enc{9}(4, 1)
2  m1?1,5
   MSC: MSG1     Eve(1) ->5 Enc{5}(4, 1)
   MSC: MSG2.1   5 ->1 Enc{1}(4, 8)
4  EveEnc2!1,4,8
3  m2!1,1
3  m2?0,1
5  m3!0,1
3  m2?1,1
4  EveEnc2?1,4,8
4  EveEnc2!1,4,8
   MSC:          Eve resending Enc{1}(4, 8)
   MSC: MSG2     Eve(9) ->1 Enc{1}(4, 8)
   MSC:          INITIATOR 1 accepted run \1 with 9
   MSC: MSG3.1   1 ->9 Enc{9}(8)
5  m3?1,5
   MSC: MSG3     Eve(1) ->5 Enc{5}(8)
   MSC:          RESPONER 5 accepted run\1 with 1
```

```
MSG1.1   IDA -> IDE Enc{IDE}(NA2, IDA)
MSG1     Eve(IDA) -> IDB Enc{IDB}(NA2, IDA)
MSG2.1   IDB -> IDA Enc{IDA}(NA2, NB2)
         Eve resending Enc{IDA}(NA2, NB2)
MSG2     Eve(IDE) -> IDA Enc{IDA}(NA2, NB2)
         INITIATOR IDA accepted run1 with IDE
MSG3.1   IDA -> IDE Enc{IDE}(NB2)
MSG3     Eve(IDA) -> IDB Enc{IDB}(NB2)
         RESPONER IDB accepted run1 with IDA
```

Fig. 8. A description of an identity misbinding attack in the *Needham–Schroeder* protocol detected with *Spin*. Model checker raw output (left) and a readable output generated by our driver (right).

Another issue about writing formulas is the labels mechanism. It should be used if possible because it avoids additional, global variables to mark a state. Labels in *Promela* are inserted into code just as in `C` language. Expression of the form $(ProcessName@LabelName)$ used in a formula, will discover a point where the process is in the labelled state.

As for the *JFKi* protocol the following two examplary formulas are presented.

$$\gamma = (JFKiProtocol@INTRUDER\_DECRYPTED\_MSG3\_LABEL \;\&\&\; cert\,!=\; IDE)$$

$$\psi = (JFKiProtocol@ACCEPTED\_INIT\_SA\_LABEL \;\&\&\; cert\,!=\; IDE \;\&\&\; secAssos == SaiE)$$

The first formula is used to detect privacy violation attack, when *Eve* decrypts the third message that was not supposed for her (global variable *cert* stores certificate of the peer chosen by the initiator and it is not IDE).

The second formula is true if the responder accepts a wrong initiators security association. That is when the association was inserted by the intruder ($SaiE$), although it should originate from a legal initiator (whose identity is kept in *cert*). There should never happen a situation when these formulas are true, so the *Büchi* automata are built for the formulas $\Box\neg\,\gamma$ and $\Box\neg\,\psi$. With analogical formulas, the ability of the adversary to change the exponentials, *nonce* and *Diffie–Hellman* group information can be checked. During the verification of *JFKi* protocol with *Spin*, none of the attacks was detected.

6. Application of the Method and Computational Results

The following model instance configuration was used for the verification results below: two parallel runs, two legal protocol users, intruder knowledge database containing two elements (*Needham–Schroeder*) or one element (*JFKi*) of every type of the complex element. In the second case, to give the adversary more abilities, any received complex element is stored in the databases nondeterministically. So the first element may not fill the queue completely. This configuration makes it feasible to verify a protocol on an average computer (AMD Athlon 2.01GHz, 2GB RAM) and lets expect the standard attacks to be detected.

Costs of example protocols verification are shown in Table 2. Our models are indicated bold. Sources of model from [**10**] are available, so scaled down to two parallel runs, they were included as a comparison. Also publicized fragments of [**15**] model give a hint of its size. At the first sight it is visible that the unminimized models present much bigger state vectors. In the case of *JFKi*, it can be very distinctly seen how beneficial for verification were the reductions of the model. The original automaton was much too complex and was only partially analyzed. The minimized model could be verified in less than a quarter of hour. Protocol security properties did hold in the *JFKi* model.

Table 2. Costs of example protocols verification.

Protocol	Time	Reached states	State vector	Used memory	Verification type
Needham–Schroeder [**10**]	2770s	3.10e+007	224b	1128.758MB	partial
Needham–Schroeder [**15**]	–	–	528b	–	code fragments
Needham–Schroeder	12.7s	3.06e+006	92b	98.094MB	full
JFKi non reduced	172s	4.00e+006	1916b	1851.350MB	partial
JFKi reduced	650s	3.62e+007	204b	1051.023MB	full

7. Conclusions and Future Plans

The proposed modelling framework has proved to be computationally *efficient*, enabling verification of more complex protocols. Although the approach

is more work consuming than using tools specialized only in verification of cryptographic protocols such as *Casper* [**21, 2**], yet it gives *more control* over the model configuration. Another fact is that the discussed method is far more readable than for example *CSP* [**2**] or clauses for the *ProVerif* program [**22**]. Hence it is more accessible for the inspection and less error prone. In addition, automatic generation of counterexamples is a true asset of the method.

The presented framework is a proposal of a method complementary to the existing ones, being aimed at solving the difficult problem of assuring correctness of safety protocols.

As for the future improvements, designing a methodology to divide the model would make it possible to verify more parallel runs of a protocol. For example, in each part the initiator would choose a different responder. Analysis of each such model would require less memory and could be possibly done concurrently on separate computers, saving the time.

A more complex task would be to create a parser that would transform a protocol specification, in a protocol description language such as *CAPSL* [**23**], into a model. The *Promela* code could be still edited by the human verifier if needed but the main work would be done automatically. This offers another opportunity, that from the same input many outputs can be generated, including several verification models or a protocol implementation [**23, 17**].

References

[1] Uk chip and pin credit / debit cards are insecure (2009) http://www.youtube.com/watch?v=JPAX321gkrw
[2] Schneider S., Ryan P., Modelling and analisis of security protocols, Addison–Wesley (2001).
[3] Holzmann G. J., Hu C., Logic Model Checking - lectures, (2008) http://spinroot.com/spin/Doc/course/
[4] Merz S., Model Checking: A Tutorial Overview, Technical report München University (2000)
[5] Mukund M., Linear-time temporal logic and Büchi automata, SPIC Mathematical Institute, Madras, India (1997)
[6] Tauriainen H., Automated testing of Büchi automata translators for linear temporal logic, Helsinki University of Technology (2000)
[7] Spin model checker: http://www.spinroot.com
[8] Spin Workshop: http://spinroot.com/spin/Workshops/index.html
[9] BEEM: BEnchmarks for Explicit Model checkers: Needham-Schroeder protocol model http://anna.fi.muni.cz/models/cgi/model_info.cgi?name=needham
[10] Khan A. S., Mukund M., Suresh S. P., Generic verification of security protocols, Springer Berlin / Heidelberg (2005)
[11] Sapiecha P., Krawczyk U., Validation of cryptographic protocols using model checker spin, CECC (2010).

[12] Lafuente A. L., Promela database: X.509 protocol model
http://www.albertolluch.com/research/promelamodels
[13] Maggi P., Sisto R., Using SPIN to to verify security properties of cryptographic protocols, In LNCS Springer-Verlag (2002): 187.
[14] Merz S., Needham–schroeder protocol model
http://www.loria.fr/ merz/papers/NeedhamSchroeder.spin
[15] Yongjian L., Rui X.,Design of a CIL Connector to Spin (2008)
[16] Compagna L., Armando A., Carbone R., LTL model checking for security protocols 23 (2009).
[17] Gordon A. D., Progress on provable implementations of security protocols, Technical Report, Microsoft Research (2009).
[18] Stamer H., Verification of cryptographic protocols, Technical Report, University of Kassel (2005).
[19] Schnoebelen Ph., The complexity of temporal logic model checking, Advances in Modal Logic (2003).
[20] Rozier K. Y., Vardi M. Y., LTL satisfiability checking (2008).
[21] Casper: A compiler for the analysis of security protocols
http://web.comlab.ox.ac.uk/people/Gavin.Lowe/Security/Casper/
[22] ProVerif: Cryptographic protocol verifier in the formal model
http://www.proverif.ens.fr/
[23] Denker G., Millen J., CAPSL and CIL Language design, Technical Report, Computer Sciene Laboratory (1999).