T U M

I N S T I T U T F Ü R I N F O R M A T I K

Automated Theorem Proving for Cryptographic Protocols with Automatic Attack Generation

Jan Jürjens and Thomas A. Kuhn

TUM-I0424
Dezember 04

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-12-I0424-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Automated Theorem Proving for Cryptographic Protocols with Automatic Attack Generation*

Jan Jürjens and Thomas A. Kuhn

Software & Systems Engineering, Dep. of Informatics, TU Munich, Germany
http://www4.in.tum.de/~{juerjens|kuhnt},{juerjens|kuhnt}@in.tum.de

Abstract. Automated theorem proving is both automatic and can be quite efficient. When using theorem proving approaches for security protocol analysis, however, the problem is often that absence of a proof of security of a protocol may give little hint as to where the security weakness lies, to enable the protocol designer to improve the protocol. For our approach to verify cryptographic protocols using automated theorem provers for first-order logic (such as e-SETHEO or SPASS), we demonstrate a method for automatically generating an attack using the automated theorem prover. The approach is general and independent from the theorem prover used. The formalization in first-order logic is intuitive and allows a mechanical translation from cryptographic protocols written as UML sequence diagrams, which we also sketch. As an example of this approach for practical security analysis of communication protocols, we examine a TLS variant protocol example.

Keywords: Security, Cryptographic Protocols, Verification, Automated Theorem Proving, Attack Generation.

1 Introduction

Automated theorem provers (ATPs) have been successfully applied to the problem of verifying cryptographic protocols for security requirements [Sch97, Wei99]. An advantage of these approaches is the potential for being not only automatic, but also quite efficient and powerful, because of the efficient decision procedures implemented in these tools and because security requirements can be formalized efficiently in first-order logic.

A disadvantage of many of these approaches is that it is often not easy to extract an attack scenario when the tool is not able to prove a protocol secure. Unlike model-checkers, for example, many automated theorem provers (such as e-SETHEO [SW00], SPASS [WBH+02], Vampire [RV01], or Waldmeister [HBVL97]) do not readily return counter-examples when theorems are found not to be provable. The reason for this is to some extent inherent: the efficiency and power of these tools relies on the fact that they perform abstract derivations, instead of explicit enumerations, and on internal optimizations. Both work against easy extraction of counter-examples. In principle, models refuting a theorem may be available, but not in a form easily readable by humans, so they would not be of much use to a security protocol developer trying to improve a protocol.

We present a new approach for verifying cryptographic protocols using automated theorem provers. The method is automatic and, for practical purposes, sufficiently efficient and powerful. Our approach translates cryptographic protocols to formulas

* Supported within the Verisoft Project of the German Ministry for Education and Research.

in first-order logic with equality (more specifically, Horn formulas). The resulting formulas are rather intuitive and compact. They are then input into any automated theorem prover supporting the TPTP input notation (such as e-SETHEO and SPASS, Vampire, and Waldmeister). This flexibility also increases the practical efficiency of the approach since one can choose the theorem prover best suited for a given verification task. If the analysis reveals that there could be an attack against the protocol, the theorem prover is again called using an automated script to produce an attack scenario. The protocol can then be improved by the designer, and the process repeated. We demonstrated our approach at the hand of a variant of the Internet protocol TLS proposed in [APS99]. Using our approach, we demonstrate an attack. We suggest a correction, and verify the corrected protocol.

As a second goal of this work, we aim to transport our results formal methods research into the context of software development by developers who may not themselves be formal methods experts. We try to achieve this by making use of a specification notation commonly used in industry. Specifically, we use UML sequence diagrams, which our approach translates to first-order logic formulas. As our verification approach is incorporated in a tool supporting automated verification of UML models for security requirements (as part of an effort presented in [Jür04]), formal verification of cryptographic protocols is available to developers without much background in formal methods but with some knowledge in UML.

Section 2 explains how to translate UML sequence diagrams to first-order logic formulas. In Section 3, we introduce our protocol case-study used in this paper, a variant of the Internet protocol TLS. Section 4 explains the transformation of first-order formulas to the ATP syntax, in which they are then analyzed in Section 5. Section 6 explains our new approach for constructing attacks from first-order logic formulas using ATPs. In Sect. 8, we explain how the translation from UML sequence diagrams to ATPs is realized in a tool. After comparing our research with related work, we close with a discussion and an outlook on ongoing research.

2 Translating UML Sequence Diagrams to First-order Logic Formulas

We assume a set **Keys** of encryption keys disjointly partitioned in sets of *symmetric* and *asymmetric* keys (where we write $\mathsf{sym}(K)$ to say that the key K is symmetric). We fix a set **Var** of *variables* and a set **Data** of *data values* (which may include *nonces* and other secrets).

The *algebra of expressions* **Exp** is the term algebra generated from the set $\mathbf{Var} \cup \mathbf{Keys} \cup \mathbf{Data}$ with the operations below. Here $\_$ denotes an argument place-holder; thus for example $\_ :: \_$ represents a binary operation. The operations are: $\_ :: \_$ (concatenation), $\mathbf{head}(\_)$ and $\mathbf{tail}(\_)$ (breaking up concatenation), $(\_)^{-1}$ (private keys), $\{\_\}\_$ (encryption), $\mathcal{D}ec\_(\_)$ (decryption), $\mathcal{S}ign\_(\_)$ (signing), and $\mathcal{E}xt\_(\_)$ (extracting from signature).

Based on this formalization of cryptographic operations, important conditions on security-critical data (such as freshness, secrecy, integrity) can then be formulated as usual following [DY83].

Our formalization below automatically derives an upper bound for the set of knowledge the adversary can gain. It analyses a security protocol specified as a UML se-

$$\begin{aligned}
\forall E_1, E_2.\ &\Big(\mathsf{knows}(E_1) \wedge \mathsf{knows}(E_2) \Rightarrow \mathsf{knows}(E_1 :: E_2) \wedge \mathsf{knows}(\{E_1\}_{E_2})\Big) \\
&\qquad\qquad \wedge \mathsf{knows}(\mathcal{S}ign_{E_2}(E_1)) \\
&\wedge\Big(\mathsf{knows}(E_1 :: E_2) \Rightarrow \mathsf{knows}(E_1) \wedge \mathsf{knows}(E_2)\Big) \\
&\wedge\Big(\mathsf{knows}(\{E_1\}_{E_2}) \wedge \mathsf{knows}(E_2^{-1}) \Rightarrow \mathsf{knows}(E_1)\Big) \\
&\wedge\Big(\mathsf{knows}(\{E_1\}_{E_2}) \wedge \mathsf{knows}(E_2) \wedge \mathsf{sym}(E_2) \Rightarrow \mathsf{knows}(E_1)\Big) \\
&\wedge\Big(\mathsf{knows}(\mathcal{S}ign_{E_2^{-1}}(E_1)) \wedge \mathsf{knows}(E_2) \Rightarrow \mathsf{knows}(E_1)\Big)
\end{aligned}$$

Fig. 1. Structural formulas

quence diagram, which for our purposes is essentially a sequence of command schemata of the form *await event e – check condition g – output event e'*.

We explain our translation from cryptographic protocols specified as UML sequence diagrams to first-order logic formulas which can be processed by the automated theorem prover e-SETHEO.

Firstly, we observe that we can eliminate some of the constructs which, to increase readability, were introduced in sequence diagrams: any usage of $\mathbf{head}(E)$ can be replaced by introducing the condition $E_1 :: E_2 = E$ and by substituting $\mathbf{head}(E)$ by E_1 at each occurrence. In a similar way, $\mathbf{tail}(\_)$, $\mathcal{D}ec\_(\_)$, and $\mathcal{E}xt\_(\_)$ can be eliminated.

The idea is then to use a predicate $\mathsf{knows}(E)$ meaning that the adversary may get to know E during the execution of the protocol. For any data value s supposed to remain confidential (which can be specified in the UMLsec model [Jür02]), one thus has to check whether one can derive $\mathsf{knows}(s)$.

The set of predicates defined to hold for a given UMLsec specification is defined as follows.

For each publicly known expression E, one defines $\mathsf{knows}(E)$ to hold.

To model the fact that the adversary may enlarge his set of knowledge by constructing new expressions from the ones he knows (including the use of encryption and decryption), one defines the formula in Fig. 1.

Suppose we are given a connection $l = (\mathsf{source}(l), \mathsf{guard}(l), \mathsf{msg}(l), \mathsf{target}(l))$ in a sequence diagram with $\mathsf{guard}(l) \equiv cond(arg_1, \ldots, arg_n)$, and $\mathsf{msg}(l) \equiv exp(arg_1, \ldots, arg_n)$, where the parameters arg_i of the guard and the message are variables which store the data values exchanged during the course of the protocol. For each such connection, we define a clause as in Fig. 2.

The formula formalisms the fact that, if the adversary knows expressions $exp_1, \ldots, exp_n$ validating the condition $cond(exp_1, \ldots, exp_n)$, then he can send them to one of the

$$\begin{aligned}
\forall exp_1, \ldots, exp_n.\ &(\mathsf{knows}(exp_1) \wedge \ldots \wedge \mathsf{knows}(exp_n) \\
&\wedge cond(exp_1, \ldots, exp_n) \\
&\Rightarrow \mathsf{knows}(exp(exp_1, \ldots, exp_n))
\end{aligned}$$

Fig. 2. Translation formulas

protocol participants to receive the message $exp(exp_1, \ldots, exp_n)$ in exchange. With this formalization, a data value s is said to be kept secret if it is not possible to derive $\mathsf{knows}(s)$ from the formulas defined by a protocol.

This way, the adversary knowledge set is approximated from above (because one abstracts away for example from the message sender and receiver identities and the message order). This means, that one will find all possible attacks, but one may also encounter "false positives". However, this treatment turns out to be rather efficient.

Note that due to the undecidability of Horn formulas with equations, one may not always be able to establish automatically that the adversary does *not* get to know a certain data value. In our practical applications of our method, this limitation has, however, not yet become observable. This is, by the way, the opposite situation for example from the method proposed in [Sch97]. We will combine the two approaches once the problem manifests itself in our approach.

3 A Variant of the Internet Protocol TLS

We will analyze a variant of the handshake protocol of TLS[1] proposed in [APS99]. To show applicability of our approach, we demonstrate a flaw, suggest a correction, and verify it. The goal of the protocol is to let a client send a secret over an untrusted communication link to a server in a way that provides secrecy and server authentication, by using symmetric session keys.

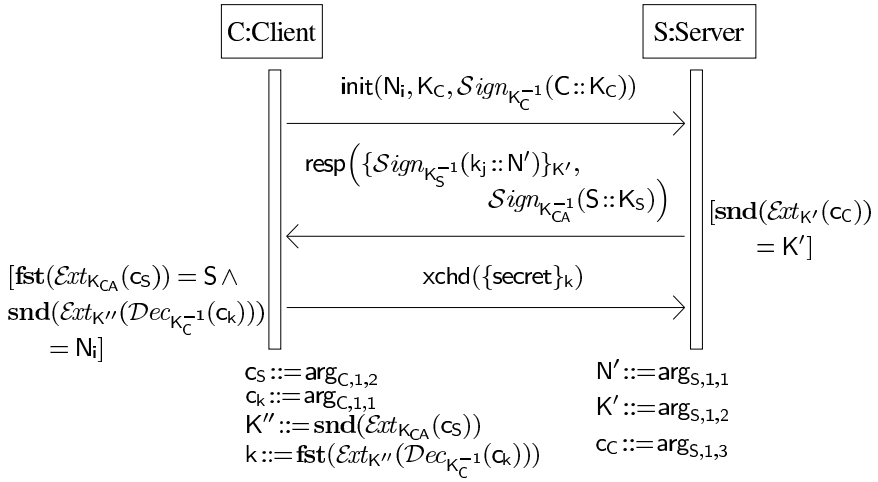

Fig. 3. Variant of the TLS handshake protocol

As shown in Fig. 3, the protocol proceeds as follows. Here we assume that the set **Var** contains elements $\mathsf{arg}_{O,l,n}$ for each $O \in \mathsf{Obj}(D)$ and numbers l and n, representing the nth argument of the operation that is supposed to be the lth operation received by O according to the sequence diagram D. The client C initiates the protocol by sending the message $\mathsf{init}(\mathsf{N_i}, \mathsf{K_C}, \mathcal{S}ign_{\mathsf{K_C}^{-1}}(\mathsf{C} :: \mathsf{K_C}))$ to the server S. If the condition $[\mathbf{snd}(\mathcal{E}xt_{\mathsf{K'}}(\mathsf{c_C}))=\mathsf{K'}]$ holds, where $K' ::= \mathsf{arg}_{S,1,2}$ and $\mathsf{c_C} ::= \mathsf{arg}_{\mathsf{S},1,3}$ (that is, the key

[1] TLS (transport layer security) is the successor of the Internet security protocol SSL (secure sockets layer).

K_C contained in the signature matches the one transmitted in the clear), S sends the message $\mathsf{resp}(\{\mathcal{S}ign_{\mathsf{K_S^{-1}}}(\mathsf{k_j} :: \mathsf{N'})\}_{\mathsf{K'}}, \mathcal{S}ign_{\mathsf{K}_{CA}^{-1}}(\mathsf{S} :: \mathsf{K_S}))$ back to C (where $\mathsf{N'} ::= \mathsf{arg_{S,1,1}}$). Then if the condition

$$[\mathbf{fst}(\mathcal{E}xt_{\mathsf{K}_{CA}}(\mathsf{c_S}))=\mathsf{S} \wedge \mathbf{snd}(\mathcal{E}xt_{\mathsf{K''}}(\mathcal{D}ec_{\mathsf{K_C^{-1}}}(\mathsf{c_k})))=\mathsf{N_i}]$$

holds, where $\mathsf{c_S} ::= \mathsf{arg_{C,1,1}}$, $\mathsf{c_k} ::= \mathsf{arg_{C,1,2}}$, and $\mathsf{K''} ::= \mathbf{snd}(\mathcal{E}xt_{\mathsf{K}_{CA}}(\mathsf{c_S}))$ (that is, the certificate is actually for S and the correct nonce is returned), C sends $\mathsf{xchd}(\{\mathsf{s_i}\}_\mathsf{k})$ to S, where $\mathsf{k} ::= \mathbf{fst}(\mathcal{E}xt_{\mathsf{K''}}(\mathcal{D}ec_{\mathsf{K_C^{-1}}}(\mathsf{c_k})))$. If any of the checks fail, the respective protocol participant stops the execution of the protocol.

4 Transformation of First-order Logic Formulas to ATP Syntax

We shortly explain how to translate the formulas from Sect. 2 into a popular automatic theorem prover input format (TPTP) used for example by the e-SETHEO prover. We explain the mapping to TPTP input format of:

- the basic intruder knowledge,
- the basic function types (enc, symenc, sign, conc) including properties,
- the first-order formula from Section 2 representing the protocol itself.

Each of these properties are expressed in TPTP input format by encapsulation within input_formula constructs. The TPTP syntax consist of the usual first-order logic symbols (written as in Fig. 4).

Syntax (TPTP)	Meaning
lowercase words	constants
uppercase letter words	variables
f(a, b, ...)	predicate function
&	conjunction
|	disjunction
=>	implication
! [Arg1, Arg2, ...] :	forall
? [Arg1, Arg2, ...] :	exists
(...)	priorities

Fig. 4. TPTP syntax

The cryptographic operations (asymmetrical and symmetrical encryption, signature) and concatenation are represented as binary functions (enc, symenc, sign, and conc) in e-SETHEO (cf. Fig. 5).

We explain the transformation to the TPTP format at the hand of the example from Sect. 3. Firstly, the initial adversary knowledge (known before the start of the protocol) is defined in Fig. 6. It is protocol-specific. The input formula which expresses the basic and in advance known intruder knowledge is expressed in Fig. 6. Here k_a and inv(k_a) represents arbitrary public and private key material of the intruder. The expression k_ca is the public key of the certification authority which is known by all parties of the protocol in advance.

The meaning of the operations is defined by input formulas representing the effect on the intruder's knowledge predicate known (cf. Fig. 7).

Meaning	UMLsec operations	e-SETHEO operations
Concatenation	_::_	conc(_,_)
Extraction	tail(_), head(_)	[derived]
Inverse keys	$(\_)^{-1}$	inv(_)
Encryption	{_}_	enc(_,_) or symenc(_,_)
Decryption	Dec_(_)	[derived]
Signing	Sign_(_)	sign(_,_)
Signature Extracting	Ext_(_)	[derived]

Fig. 5. UMLsec vs. e-SETHEO operations

```
input_formula(previous_knowledge,axiom,(
    (   knows(k_a)
      & knows(inv(k_a))
      & knows(k_ca) )   )).
```

Fig. 6. Attackers initial knowledge

The main part of the transformation to the e-SETHEO approach is the protocol definition itself (cf. Fig. 8). The input formula expresses the first-order formula as mentioned in the above UMLsec transformation section. The protocol itself is expressed by a for-all quantification over the pieces of messages which are transferred over the communication channel (e.g. network). The message variables ArgC_11 and ArgC_12 stand for the messages received by the client. The message variables ArgS_11, ArgS_12 and ArgS_13 stand for the server receiving messages parts. The protocol example includes three messages (cf. Fig. 3). The first one sent from the client, the second one from the server and the third one sent again from the client. Each message is expressed by a single clause of the main conjunction. Therefore three clauses occur in the example. The first one

$$\mathsf{knows(n)}\ \&\ \mathsf{knows(k\_c)}\ \&\ \mathsf{knows(sign(conc(c,k\_c),\ inv(k\_c))}\ ))$$

is the message sent from the client to the server. It has no preconditions because of the initial message type. The second and third protocol messages are specified by implications. In the preconditions of the implications the expected pieces of input message are expressed by known(Arg_C11), known(ArgC_12), and known(ArgS_11), known(ArgS_12), known(ArgS_13) respectively. Conditional parts of the received messages are expressed by equations. For instance, the checking if the key of the signed message equals the transmitted key k_c is expressed by:

$$?\ [X]\ :\ \mathsf{equal(\ sign(conc(X,\ ArgS\_12),\ inv(ArgS\_12)\ ),\ ArgS\_13\ )}$$

The conditional equations use the binary function equal(a, b) which is a predefined expression of TPTP syntax and represents the equality relation. In order to express extractions of parts of the messages a pattern matching approach is chosen. One example is the third line of the third protocol message:

$$\mathsf{equal(sign(conc(s,\ DataC\_KK),\ inv(k\_ca)),\ ArgC\_12\ )}$$

The value which is determined by pattern matching is DataC_KK the server's public key for the signature verification. The implications postconditions include the messages send over the communication channel like for example

$$\mathsf{knows(symenc(secret,\ DataC\_k))}$$

```
%---- Asymmetrical Encryption ----
input_formula(enc_equation,axiom,(
 ! [E1,E2] :
   ( (   knows(enc(E1, E2))
       & knows(inv(E2)) )
    =>   knows(E1) )  )).

%------ Symmetrical Encryption ------
input_formula(symenc_equation,axiom,(
 ! [E1,E2] :
   ( (   knows(symenc(E1, E2))
       & knows(E2) )
    =>   knows(E1) )  )).

%------------ Signature -----------
input_formula(sign_equation,axiom,(
 ! [E,K] :
   ( (   knows(sign(E, inv(K) ) )
       & knows(K) )
    =>   knows(E) )  )).
```

```
%----- Basic Relations on Knowledge -----
input_formula(construct_message_1,axiom,(
 ! [E1,E2] :
   ( (     knows(E1)
         & knows(E2) )
    => (   knows(conc(E1, E2))
         & knows(enc(E1, E2))
         & knows(symenc(E1, E2))
         & knows(sign(E1, E2)) ) )  )).

input_formula(construct_message_2,axiom,(
 ! [E1,E2] :
   (       knows(conc(E1, E2))
    => (   knows(E1)
         & knows(E2) ) )  )).
```

Fig. 7. General ATP rules

in the last message from the client. The specification of fresh keys as it is used in the second message from the server is expressed by encapsulating it within an unary function kgen. For instance the key from the first message of the client within an unary function named kgen, e.g. kgen(ArgS_12).

Other security protocols can be transformed through a converter program automatically.

The logical transformation of the security protocol can be verified against an conjecture statement like depicted in figure 9. Here an confidentiality property is tested. In other words the theorem prover proofs if the knowledge set of the intruder contains the secret information or that the information can be derived from other knowledge set information of the intruder.

5 Protocol Analyses with ATPs like e-SETHEO

The prover SETHEO (SEquential THEOrem prover)[2] is an efficient automated theorem prover for first oder logic in clausal normal form [Sch01]. Based on the model elimination calculus it has been developed by the Automated Reasoning Group at the Technische Universität München. To handle and reasoning about equality properties within the analysis of protocols the SETHEO prover has been extended to the e-SETHEO prover. The e-SETHEO prover includes further and progressive techniques to handle equality efficiently. The two main techniques are unfailing completion [BDP89] and superposition and to a small extent lazy para-modulation [GRS87].

We use of e-SETHEO for verifying security protocols as a "black-box": A TPTP input file is presented to the ATP and an output from the ATP is observed. No internal properties of or information from e-SETHEO is used. This allows one to use

[2] Other ATPs like the SPASS prover can also be used for the analysis and have almost the same calling properties like SETHEO.

```
input_formula(tls_abstract_protocol,axiom,(
![ArgS_11, ArgS_12, ArgS_13, ArgC_11, ArgC_12] : (
  ![DataC_KK, DataC_k, DataC_n] : (
     % Client -> Attacker (1. message)
     (   knows(n)
       & knows(k_c)
         % asymetrical public key
       & knows(sign(conc(c, k_c), inv(k_c) ) )
         % signature of concatination
     )
   & % Server -> Attacker (2. message)
     (   (  knows(ArgS_11)
            % first part of receiving server message
          & knows(ArgS_12)
            % second part of receiving server message
          & knows(ArgS_13)
            % third part of receiving server message
          & ( ? [X] : equal( sign(conc(X, ArgS_12), inv(ArgS_12) ), ArgS_13 ) )
            % condition test of snd(...)
         )
      => (  knows(enc(sign(conc(kgen(ArgS_12), ArgS_11), inv(k_s) ), ArgS_12 ) )
          & knows(sign(conc(s, k_s), inv(k_ca) ) ) )
         )
   & % Client -> Attacker (3. message)
     (   (  knows(ArgC_11)
          & knows(ArgC_12)
          & equal(sign(conc(s, DataC_KK), inv(k_ca)), ArgC_12 )
            % K'' = snd(Ext_{K_CA}(arg_C_12)
          & equal(enc(   sign(conc(DataC_k, DataC_n), inv(DataC_KK) ),
            k_c), ArgC_11 )
            % Decryption of message Dec_{K_C-1}(arg_C_11))
          & ( ? [DataC_ks] : equal(sign(conc(s, DataC_ks), inv(k_ca)), ArgC_12 ) )
            % First Condition Testing == [fst(...)=S]
          & equal(enc(sign(conc(DataC_k, n), inv(DataC_KK) ), k_c), ArgC_11 )
            % Second Condition Testing == [snd(Ext_K''(...))=N']
         )
      => (  knows(symenc(secret, DataC_k)) )
     )
) )  )).
```

Fig. 8. Protocol Specification

```
% ------------------------ Attack -----------------
% Here you can see that the secret can be captured
% by an attacker!

input_formula(attack,conjecture,( knows(secret)  )).
```

Fig. 9. Example of a conjecture statement (security property)

e-SETHEO interchangeable with any other ATP accepting TPTP as an input format (such as SPASS, Vampire and Waldmeister) when it may seem fit.

The running of e-SETHEO involves few parameters:

```
run-e-setheo [--add-eq] <input file> <resources> <strategy>
```

The add − eq option is used to enable the automatic adding of the equality properties like symmetricity and reflexivity to the TPTP file. The amount of time resources

in seconds of e-SETHEO can be limited by the resources parameter, e.g. 300 seconds. In the e-SETHEO approach several prove strategies are used in parallel within one run. The time resources are subdivided to the individual prover strategies e-SETHEO uses. In order to tell the proof system to use only a special one the strategy parameter can be used (cf. [LS01]). The analysis of the TLS variant protocol is done by using the e-SETHEO call:[3]

```
run-e-setheo tlsvariant-freshkey-check.tptp 300
```

An extract of the output of e-SETHEO is shown in Fig. 10. The result can be seen in the bottom part of the output where the prover returns Proof found which means that the conjection known(secret) can be derived from the defined rules within three seconds by one the provers contained in e-SETHEO.

```
E-SETHEO csp03 single processor running on host ...
(c) 2003 Max-Planck-Institut fuer Informatik and
    Technische Universitaet Muenchen

tlsvariant-freshkey-check.tptp
...
time limit information: 300 total (entering statistics module).
problem analysis ...
testing if first-order ...
first-order problem
...
statistics: 19 0 7 46 3 6 2 0 1 2 14 8 0 2 28 6
...
schedule selection: problem is horn with equality (class he).
schedule:605 3 300 597
...
entering next strategy 605 with resource 3 seconds.
...
analyzing results ...
proof found
time limit information: 298 total / 297 strategy (leaving wrapper).
...
e-SETHEO done. exiting
```

Fig. 10. Output of e-SETHEO

6 Attack Tracking Approach

We describe the approach of attack tracking with the assistance of an automatic theorem prover (e.g. e-SETHEO or SPASS). We give an overview of the idea, describe the tool implementation, and show on the TLS variant protocol how flaws can be found. We fix the flaw and again verify the correctness of the protocol using the automatic theorem prover.

In the protocol body (cf. Fig. 8) we can see that the knowledge of an attacker can influence the running of the protocol. For instance, the attacker can use his own

[3] The input file is available upon request from the authors and is documented in the appendix of the longer version of the paper.

key material k_a and inv(k_a) in order to fake messages so that the attacker can gain additional knowledge from the other protocol participants.

As mentioned in the last section, the protocol specification together with a conjecture about the possible attacker knowledge can be proved by the automatic theorem prover. Until the ATP returns the message proof found, this means that an (partial) assignment exists for the message variables. In the TLS variant protocol example, this means that an assignment to the message variables ArgC_11, ArgC_12, ArgS_11, ArgS_12, ArgS_13 can be found by an ATP proof of the conjecture (such as known(secret)). The automatic theorem prover e-SETHEO does not return such an assignment explicitly because the proof technique operates on an abstract level. In order to find such an assignment, the message variables have to be filled step-by-step in a proper way by possible values like k_a, n, k_c, enc(n, k_a),..., so that e-SETHEO continues to find a proof until all assignment values are found. The first value found for the first message variable is fixed to that message variable and used during the search for the other message variables. Step by step the message variables are replaced by values for which the invariant that the automatic theorem prover finds a proof holds. E.g. if in the TLS variant protocol the first message variable ArgC_11 is replaced by the value n, the automatic theorem prover e-SETHEO returns Proof found. The resulting protocol reduces then to only four message variables ArgC_12, ArgS_11, ArgS_12, and ArgS_13. Therefore, the number of message variables in the for-all quantification part of the protocol is reduced until all message variables have been instantiated. A generalization of this idea leads to the following algorithm A:

Let protocol($\mathsf{Arg}_1, \mathsf{Arg}_2, ..., \mathsf{Arg}_n$) $\in \{\mathsf{true}, \mathsf{false}, -\}$ with the meaning that a protocol is checked true, false or unknown by an automatic theorem prover (e.g. e-SETHEO) by replacing the message variables occurring in the protocol's for-all quantification part by values of the assignment.

```
Algorithm A:
Input:  VAL = set of message values val1, val2, ...
Output: Assignment for the message variables ARG = { Arg1, ..., Argn}

Assignment[1] = Assignment[2] = ... = '-';
FOREACH arg in ARG DO
  FOREACH val in VAL DO
    IF protocol(Assignment) = true THEN
      Assignment[arg] = val;
```

After the algorithm A terminates on the protocol it gives back an assignment of the message variables where a value has been found (assuming that a value was included in the set VAL). A protocol attacking assignment is found if for all message variables a value has been found.

The reason why the above attack tracking algorithm works, is that the protocol function has a homomorphic property[4] of the form:

$$\forall VAL, ARG : \text{protocol}(val_1, ..., val_i, ..., val_n) = \text{protocol}(val_1, Arg_2, ..., Arg_i, ...Arg_n) \wedge ... \wedge$$

[4] In other words the assignment of the message variables does not depend on the order of assigning by the values.

$$\text{protocol}(Arg_1, ..., Arg_{i-1}, val_i, Arg_{i+1}, ...Arg_n) \wedge ... \wedge$$
$$\text{protocol}(Arg_1, ..., Arg_i, ..., Arg_{n-1}, val_n)$$

where $\wedge$ is the well-known conjunction operation with the exception that the occurrence of the assignment ' $-$' is ignored by the boolean conjunction operation, e.g. true $\wedge$ - $\wedge$ false = false.

There are several advantages of the above attack tracking approach:

- The run time complexity of the algorithm A needs linear many calls of the ATP system in relation to the input value set. Here in general the input value set consists out of atomic values like n, k_c, ... or is a composition of the protocol operations enc, symenc, sign, and conc assigned with atomic values. The set VAL is infinite but can be reduced to a finite set in practice, e.g. only compositions to a constant depth are relevant for the protocol.
- If the person who analyses the protocol has some pre-knowledge about the attack, the set VAL can be made even smaller or can be a set constructed by deletion of invalid or useless values pre-knowledge in-cooperated.
- Security protocols guarantee the homomorphic property mentioned above.
- Even a partial assignment of values to message variables can be found which can give the expert user a hint for further attack investigations.

The above mentioned algorithm A has been implemented by a perl script environment. Fig. 11 gives an overview of the attack tracking perl tool environment.

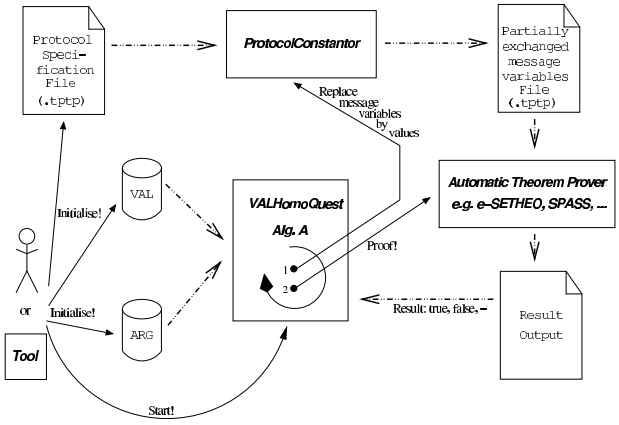

Fig. 11. Attack tracking tools suite

The set VAR and ARG are represented by two file stores which are initialized by the user directly or via the output of a tool, e.g. UMLsec tool suite or a Message Generator Tool. The security protocol specification (TPTP) is also generated by the user himself or by a tool. This generation is done in the same way as described in Sect. 2 and 4 above. The user then starts the VALHomoQuest perl script[5] with the following parameter syntax:

```
VALHomoQuest <VAL filename> <ARG filename> <protocol filename> <time>
```

[5] Available via http://www4.in.tum.de/~kuhnt

The VAL and ARG text filename is the filename where the values and message variables are listed. Each text line corresponds one value or message variables. The protocol filename corresponds to the protocol specification as a TPTP input file. The time parameter is used for telling the connected automated theorem prover how much time/space resources it is allowed to use for each proofing trial. After consuming all the time resources the automated theorem prover gives back ' −' which means no answer was found for the time limit offered.

The attack tracking tool suite is used on the TLS variant protocol in order to demonstrate the method. The protocol is given to the VALHomoQuest perl script where as protocol filename tlsvariant − freshkey − check.tptp, the time resources for the e-SETHEO automated theorem prover is set to 300 seconds, the VAL file is constructed in this example by hand from the user[6] as depicted in Fig. 12, and the ARG file looks like depicted in Fig. 13:

```
k_a
inv(k_a)
k_ca
n
k_c
sign(n,k_a)
sign(conc(c, k_a),inv(k_a))
enc(sign(conc(kgen(k_a),n),inv(k_s)),k_a)
enc(sign(conc(kgen(k_a),n),inv(k_s)),k_c)
enc(sign(conc(kgen(k_a),n),k_s),k_a)
sign(conc(s, k_s), inv(k_ca) )
```

Fig. 12. Values for attack (VAL)

```
ArgS_11
ArgS_12
ArgS_13
ArgC_11
ArgC_12
```

Fig. 13. Message Variables (ARG)

The command is then invoked by:

```
VALHomeQuest VAL.dat ARG.dat tlsvariant-freshkey-check.tptp 300
```

While proceeding the analysis the tool gives back the current working parameters for status purposes and after finishing the analysis the assignment to the message variables are given back (cf. Fig. 14. For this example the attack tracking tool needs around 20 seconds on a SunFire 3800 (4 processors, 6 GByte RAM, Solaris 9).

```
VHQ: Attack Assignment:
  - ArgS_11:n
  - ArgS_12:k_a
  - ArgS_13:sign(conc(c, k_a),inv(k_a))
  - ArgC_11:enc(sign(conc(kgen(k_a),n),inv(k_s)),k_c)
  - ArgC_12:sign(conc(s, k_s), inv(k_ca) )
```

Fig. 14. Attack Assignment

This means that the protocol does not provide its intended security requirement, secrecy of secret, against a realistic adversary.

[6] In general this VAL file is generated automatically by an Message Generator or imported from a connected other tool like the UMLsec tool suite

The derived message flow diagram corresponding to a man-in-the-middle attack depicted in Fig. 15.

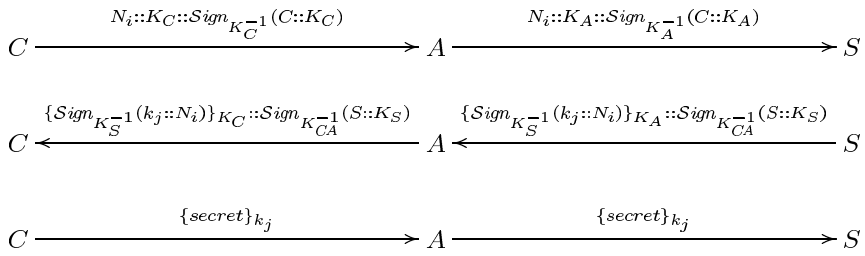

Fig. 15. Attack Visualization: Man-in-the-middle

We propose to change the protocol to get a new specification by substituting $k_j :: N_i$ in the second message (server to client) by $k_j :: N_i :: K_C$ and by including a check regarding this new message part at the client. Here, the public key K_C of C is included representatively for the identity of C (in fact, one could also use $k_j :: N_i :: C$ instead).

Again, in traditional informal notation, the modified protocol line would be written as follows:

$$S \rightarrow C : \{\mathcal{S}ign_{\mathsf{K_S^{-1}}}(\mathsf{k_j :: N_i :: K_C})\}_{\mathsf{K_C}}, \mathcal{S}ign_{\mathsf{K_{CA}^{-1}}}(\mathsf{S :: K_S})$$

Now the new version with the additional signature information about the client key k_c can be verified by the automated theorem prover approach. The calling of e-SETHEO looks like this:

```
run-e-setheo tlsvariant-fix-freshkey-check.tptp 300
```

When e-SETHEO runs on the fixed version of the protocol it now gives back Model found within 5 seconds which means that there exists no proof of the conjecture known(secret) and therefore the attacker cannot gain the secret knowledge anymore.

Note that since our approach approximates the adversary knowledge set from above (which is a reason for its efficiency), it could consider protocols to be insecure which are in fact secure. However, this problem has not shown up in any practical protocols. If it would, the designer would, using the attack generation facility presented in this section, see that the scenario that is produced is not a realistic attack.

7 Related Work

There has been extensive research in using formal models to verify secure systems. Their main areas of application in this domain include security policies and security protocols. Early influential work was given in [DY83, MCF87, BAN89, Mea91, Syv91]. In particular, [Mea91] is based on the logic programming language Prolog and thus related to using an automated theorem prover. However, there are differences, for example in that [Mea91] uses a state search approach (see [Mea95] for details) whereas our approach abstractly bounds the possible adversary knowledge from above using

inference rules of first order logic. Authentication logics such as [BAN89, Syv91] have in common with our approach the abstract way of reasoning, which often allows automated verification. Our approach differs from some of these approaches by also automatically finding attacks. Several approaches deal specifically with confidentiality requirements, as we do here (for example [Fol94]). In a line of work started at [THG98], the strand space approach provides a formal method tailored to the analysis of cryptographic protocols. Its aim is to provide a verification approach which is sufficiently simple to be used manually. While we believe that our approach for formalizing cryptographic protocols in first-order logic is also quite intuitive, our aim is to provide an automatic method to enable its use (through the UML notation) also by developers without much mathematical background. The process algebra CSP has been employed quite successfully, for example in [Sch96, LR97, HPS01]. This has been done to a large extent by making use of the model-checker FDR. Although, as a model-checker, prone to the state-explosion problem, its use has been rather successful due to the maturity of the tool and to results allowing the user to restrict the search space. An alternative approach here is the "rank function" method which is being implemented in an independent tool [HS00].

Most closely related to our work are approaches that also use first-order logic. [Sch, Wei99, Coh00] employ first-order reasoning is for proving security properties of cryptographic protocols using various automated theorem provers.

Relatively little work so far has been done using automated theorem provers for cryptographic protocol analysis. As an example, [Sch97] uses e-SETHEO for verifying authentication protocols in an approach based on BAN logic [BAN89]. In contrast to that work, our approach can also deal with secrecy properties, and can generate attacks automatically. [Wei99] uses the ATP SPASS to analyze the Neuman-Stubblebine protocol. Again, an advantage of our approach is that it can also provide attack scenarios. As another use of ATPs, [GP00] uses the Otter tool for validation of cryptographic protocols by efficient automated testing.

[GSG99, Gol03, Mea03] give short overviews and point out open problems and critical things. For an overview of the work on verifying security protocols with a focus on the process algebra CSP, see [RSG+01]. See [SC01] for an overview on authentication logics.

Other work in security engineering using UML includes [VML03], presenting a business process-driven framework for security engineering with the UML. It is motivated by the fact that security requirements can be communicated best between customers and developers at the level of business process models in UML. [LOT03] uses the notation SDL for a formal analysis of security systems. The approach uses a specification notation based on HMSC/MSC, which can be automatically translated into a generic SDL specification. [HH03] uses UML to specify information flow requirements for Java-based systems.

8 Tool Integration

For the ideas that were presented in the previous sections to be of benefit in practice, it is important to have advanced tool-support to assist in using them, which is sketched in Fig. 16. [7]

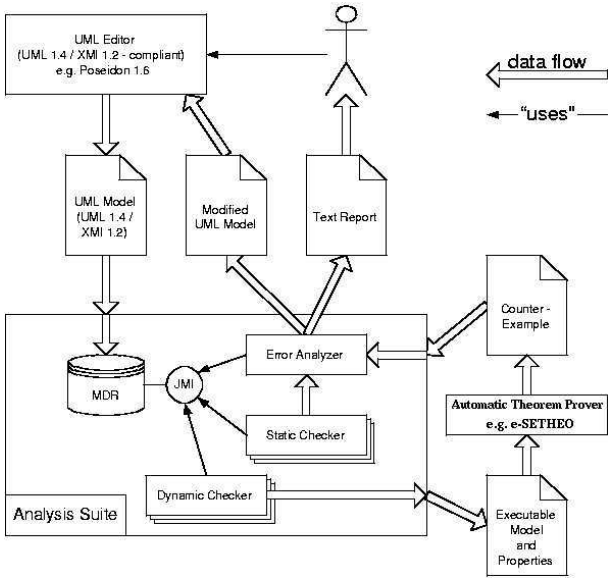

Fig. 16. UML tools suite

9 Conclusion and Future Work

We presented a new approach for verifying cryptographic protocols, given as UML sequence diagrams, using automated theorem provers. The method is automatic and, for practical purposes, sufficiently efficient and powerful. Our approach translates UML sequence diagrams to formulas in first-order logic with equality (more specifically, Horn formulas). The resulting formulas are rather intuitive and compact. They are then input into any automated theorem prover supporting the TPTP input notation. If the analysis reveals that there could be an attack against the protocol, the theorem prover is again called using an automated script to produce an attack scenario. The protocol can then be corrected by the designer, and the process repeated. We demonstrated our approach at the hand of a variant of the Internet protocol TLS proposed in [APS99]. Using our approach, we demonstrated an attack. We suggested an improved version, and verified the corrected protocol.

A feature of our work is that it combines an industrially used notation such as UML with the formal analysis of cryptographic protocols using automated theorem proving. It is both automatic and efficient, and compared with other automated theorem proving approaches, it also features an automated attack generation. It does not rely on especially implemented algorithms but may use a number of well-developed and

[7] The tool is available for general use through a web-interface (User: Tooluser, Password: Imayusethis, Link: http://www4.in.tum.de/csduml/interface/interface.html

well-supported tools interchangable, and profit from its improvements in the future without the necessity of any adjustments.

Since automated theorem provers for first-order logic with equality rely on semi-decision procedures, a limitation of our approach in principle could be that for a given protocol, the theorem prover can neither establish existence nor absence of attacks. However, in our tests, this theoretical problem has not become apparent. Also, this seems to be a problem which is not specific to our approach but of a more general nature [DLMS99]. Since one can use our method invariably with any theorem prover accepting the TPTP notation, facing such a problem, one could then easily use another prover with different decision procedures. Also, since our approach approximates the adversary knowledge set from above (which is a reason for its efficiency), it could consider protocols to be insecure which are in fact secure. Again, this problem has not shown up in any practical protocols. If it would, the designer would see that the scenario that is produced it is not a realistic attack.

As our verification approach is incorporated in a UML tool, it makes formal verification of cryptographic protocols available to developers without much background in formal methods but with some knowledge in UML.

The method proposed here is being applied in security-relevant projects with industry, for example in projects of a German car manufacturer, and a telecommunications company.

For future work, we aim to apply the attack tracking approach to other protocols. The construction of the message set VAL in Sect. 6 could be improved, for example with the usage of intelligent strategies for generation of relevant terms where support of additional user interaction is provided.

Acknowledgments Helpful discussions with Gernot Stenz, Robert Schmidt, Johann Schumann, and Bo Zhang are gratefully acknowledged. We thank Guido Wimmel and Gernot Stenz for their comments on a previous version of the paper.

References

[APS99] V. Apostolopoulos, V. Peris, and D. Saha. Transport layer security: How much does it really cost? In *Conference on Computer Communications (IEEE Infocom)*, pages 717–725. IEEE Computer Society, March 1999.

[BAN89] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989.

[BDP89] L. Bachmair, N. Dershowitz, and D. Plaisted. Completion without failure. In H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures, Volume 2: Rewriting Techniques*, pages 1–30. Academic Press, 1989.

[Coh00] Ernie Cohen. Taps: A first-order verifier for cryptographic protocols. In *13th IEEE Computer Security Foundations Workshop — CSFW'00*, pages 144–158, Cambridge, UK, 3–5 July 2000. IEEE Computer Society Press.

[DLMS99] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In *Workshop on Formal Methods and Security Protocols (FMSP'99)*, 1999.

[DY83] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(2):198–208, 1983.

[Fol94] S. Foley. Reasoning about confidentiality requirements. In *CSFW 1994*, pages 150–160. IEEE Computer Society, 1994.

[Gol03] Dieter Gollmann. Analysing security protocols. In *First International Conference, FASec 2002*, volume 2629 of *LNCS*, pages 71–80, 2003.

[GP00] S. Gürgens and R. Peralta. Validation of cryptographic protocols by efficient automated testing. In *FLAIRS 2000*, pages 7–12. AAAI Press, May 2000.

[GRS87] Jean H. Gallier, Stan Raatz, and Wayne Snyder. Theorem proving using rigid E-unification equational matings. In *Proceedings, Symposium on Logic in Computer Science*, pages 338–346, Ithaca, New York, 22–25 June 1987. The Computer Society of the IEEE.

[GSG99] S. Gritzalis, D. Spinellis, and P. Georgiadis. Security protocols over open networks and distributed systems: Formal methods for their analysis, design, and verification. *Computer Communications Journal*, 22(8):695–707, 1999.

[HBVL97] T. Hillenbrand, A. Buch, R. Vogt, and B. Lochner. WALDMEISTER - high-performance equational deduction. *Journal of Automated Reasoning*, 1997.

[HH03] R. Heldal and F. Hultin. Bridging model-based and language-based security. In E. Snekkenes and D. Gollmann, editors, *Computer Security - ESORICS 2003, 8th European Symposium on Research in Computer Security*, volume 2808 of *LNCS*, pages 235–252, Gjøvik, Norway, October 13–15 2003. Springer.

[HPS01] M. Heisel, A. Pfitzmann, and T. Santen. Confidentiality-preserving refinement. In *IEEE Computer Security Foundations Workshop*, pages 295–306. IEEE Computer Society, 2001.

[HS00] J. Heather and S. Schneider. Towards automatic verification of authentication protocols on an unbounded network. In *Computer Security Foundations Workshop 13*, 2000.

[Jür02] J. Jürjens. UMLsec: Extending UML for secure systems development. In J.-M. Jézéquel, H. Hußmann, and S. Cook, editors, *UML 2002 – The Unified Modeling Language*, volume 2460 of *LNCS*, pages 412–425. Springer, September 30 – October 4, 2002. 5th International Conference.

[Jür04] J. Jürjens. *Secure Systems Development with UML*. Springer, March 2004. To be published.

[LOT03] J. Lopez, J. Ortega, and J. Troya. Applying sdl to formal analysis of security systems. In R. Reed and J. Reed, editors, *SDL 2003: System Design, 11th International SDL Forum*, volume 2708 of *LNCS*, pages 300–316, Stuttgart, Germany, July 1-4 2003. Springer.

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

[LS01] R. Letz and G. Stenz. *Model Elimination and Connection Tableau Procedures*, pages 2015–2114. Elsevier Science/MIT press, 2001.

[MCF87] J.K. Millen, S.C. Clark, and S.B. Freedman. The Interrogator: Protocol security analysis. *IEEE Transactions on Software Engineering*, SE-13(2):274–288, February 1987.

[Mea91] C. Meadows. A system for the specification and analysis of key management protocols. In *IEEE Symposium on Security and Privacy*, pages 182–195, 1991.

[Mea95] C. Meadows. The nrl protocol analyzer: An overview. *Journal of Logic Programming*, 1995.

[Mea03] C. Meadows. What makes a cryptographic protocol secure? the evolution of requirements specification in formal cryptographic protocol analysis. In P. Degano, editor, *Programming Languages and Systems, 12th European Symposium on Programming (ESOP 2003)*, volume 2618 of *LNCS*, pages 10–21, Warsaw, Poland, April 7–11 2003. Springer.

[RSG+01] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *The Modelling and Analysis of Security Protocols: the CSP Approach*. Addison-Wesley, Reading, MA, 2001.

[RV01] A. Riazanov and A. Voronkov. Vampire 1.1 (system description). In R. Goré, A. Leitsch, and T. Nipkow, editors, *1st Intl. Joint Conf. Automated Reasoning*, volume 2083 of *LNCS*, pages 376–380, Siena, Italy, June 2001. Springer.

[SC01] P. F. Syverson and I. Cervesato. The logic of authentication protocols. In *Foundations of Security Analysis and Design (FOSAD 2000)*, volume 2171 of *LNCS*, pages 63–136, 2001.

[Sch] J. Schumann. PIL: A tool for the automatic analysis of authentication protocols. pages 500–504.

[Sch96] S. Schneider. Security properties and CSP. In *IEEE Symposium on Security and Privacy*, pages 174–187, 1996.

[Sch97] J. Schumann. Automatic verification of cryptographic protocols with SETHEO. In W. McCune, editor, *CADE-14*, volume 1249 of *LNCS*, pages 87–100. Springer, 1997.

[Sch01] Johann M. Schumann. *Automated Theorem Proving in Software Engineering*. Springer, 2001.

[SW00] G. Stenz and A. Wolf. E-SETHEO: An Automated^3 Theorem Prover – System Abstract. In R. Dyckhoff, editor, *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX-2000)*, volume 1847 of *LNAI*, pages 436–440, St. Andrews, Scotland, July 2000. Springer.

[Syv91] P. Syverson. The use of logic in the analysis of cryptographic protocols,. In *1991 IEEE Symposium on Research in Security and Privacy*, Oakland, CA, May 1991. IEEE Computer Society.

[THG98] F. Javier Thayer Fábrega, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Why is a security protocol correct? In *Proceedings 1998 IEEE Symposium on Security and Privacy*, pages 160–171, May 1998.

[VML03] J. Vivas, J. Montenegro, and J. Lopez. Towards a business process-driven framework for security engineering with the UML. In C. Boyd and W. Mao, editors, *6th International Conference on Information Security (ISC 2003)*, volume 2851 of *LNCS*, pages 381–395, Bristol, UK, October 1-3 2003. Springer.

[WBH+02] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topić. Spass version 2.0. In A. Voronkov, editor, *18th International Conference on Automated Deduction*, volume 2392 of *LNAI*. Springer, 2002.

[Wei99] C. Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In H. Ganzinger, editor, *16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *LNCS*, pages 314–328. Springer, 1999.

10 Appendix

10.1 tlsvariant-freshkey-check.tptp

```
%------------------------------------------------------------------------
% File     : The basic (-without- fix) TLS Variant Protocol is modelled here!
%            (IEEE Infocom 99) [Authors: Thomas Kuhn, Jan Jrjens]
% Domain   :
% Problem  :
% Version  :
% English  :

% Refs     :
% Source   :
% Names    :
% Status   : unknown
% Rating   : ?
% Syntax   : Number of formulae    :   26 (   6 unit)
%            Number of atoms       :   65 (  40 equality)
%            Maximal formula depth :   14 (   4 average)
%            Number of connectives :   40 (   1 ~  ;   0  |;  19  &)
%                                         (   0 <=>;  20 =>;   0 <=)
%                                         (   0 <~>;   0 ~|;   0 ~&)
%            Number of predicates  :    2 (   0 propositional; 1-2 arity)
%            Number of functors    :   16 (   8 constant; 0-2 arity)
%            Number of variables   :   60 (   0 singleton;  60 !;   0 ?)
%            Maximal term depth    :    6 (   1 average)

% Comments :
%          : tptp2X -f tptp -t add_equality tlsvariant.p
%------------------------------------------------------------------------

%----------------------- Asymmetrical Encryption ------------------------
input_formula(enc_equation,axiom,(
    ! [E1,E2] :
      ( (   knows(enc(E1, E2))
          & knows(inv(E2)) )
        => knows(E1) )  )).

%----------------------- Symmetrical Encryption -------------------------
input_formula(symenc_equation,axiom,(
    ! [E1,E2] :
      ( (   knows(symenc(E1, E2))
          & knows(E2) )
        => knows(E1) )  )).

%-------------------------- Signature -----------------------------------
input_formula(sign_equation,axiom,(
    ! [E,K] :
      ( (   knows(sign(E, inv(K) ) )
```

```
          & knows(K) )
        => knows(E) )  )).

%---- Basic Relations on Knowledge where conc, enc, symenc and sign is included ----
input_formula(construct_message_1,axiom,(
    ! [E1,E2] :
      ( (   knows(E1)
          & knows(E2) )
        => (   knows(conc(E1, E2))
             & knows(enc(E1, E2))
             & knows(symenc(E1, E2))
             & knows(sign(E1, E2)) ) )  )).

input_formula(construct_message_2,axiom,(
    ! [E1,E2] :
      ( ( knows(conc(E1, E2)) )
        => ( knows(E1)
           & knows(E2) ) )  )).

%---------------------- Attackers Initial Knowledge ----------------------
input_formula(previous_knowledge,axiom,(
    (   knows(k_a)
      & knows(inv(k_a))
      & knows(k_ca)
    )   )).

%--------------------- TLS Main Protocol Specification --------------------
input_formula(tls_abstract_protocol,axiom,(
%PERLPROTOCOLCHECK_BEGINNING
! [ArgS_11, ArgS_12, ArgS_13, ArgC_11, ArgC_12] :
  (
    ![DataC_KK, DataC_k, DataC_n] :
       (   % Client -> Attacker (1. message)
           (   knows(n)
             & knows(k_c)                              % asymetrical public key
             & knows(sign(conc(c, k_c), inv(k_c) ) )   % signature of concatination
           )
         & % Server -> Attacker (2. message)
           (    (   knows(ArgS_11)  % first part of receiving server message
                  & knows(ArgS_12)  % second part of receiving server message
                  & knows(ArgS_13)  % third part of receiving server message
                  & ( ? [X] : equal( sign(conc(X, ArgS_12), inv(ArgS_12) ),
                      ArgS_13 ) ) % condition test of snd(...)
                )
             => (   knows(enc(sign(conc(kgen(ArgS_12), ArgS_11), inv(k_s) ), ArgS_12 ) )
                  & knows(sign(conc(s, k_s), inv(k_ca) ) ) )
           )
         & % Client -> Attacker (3. message)
           (    (   knows(ArgC_11)
                  & knows(ArgC_12)
                  & equal(sign(conc(s, DataC_KK), inv(k_ca)), ArgC_12 )
                    % K'' = snd(Ext_{K_CA}(arg_C_12)
                  & equal(enc(   sign(conc(DataC_k, DataC_n), inv(DataC_KK) ),
  k_c), ArgC_11 )
                    % Decryption of message Dec_{K_C-1}(arg_C_11))
                  & ( ? [DataC_ks] : equal(sign(conc(s, DataC_ks), inv(k_ca)
  ), ArgC_12 ) )
                    % First Condition Testing == [fst(...)=S]
                  & equal(enc(sign(conc(DataC_k, n), inv(DataC_KK) ), k_c),
  ArgC_11 )
                    % Second Condition Testing == [snd(Ext_K''(...))=N']
                )
             => ( knows(symenc(secret, DataC_k)) )
           )
       )
  )
)).
```

```
%PERLPROTOCOLCHECK_ENDING

% ----------------------- Attack ----------------------------------------
% Here you can see that the secret can be captured by an attacker!

input_formula(attack,conjecture,(
    knows(secret)  )).
```

10.2 tlsvariant-fix-freshkey-check.tptp

```
%------------------------------------------------------------------------
% File     : The basic (-with- fix) TLS Variant Protocol is modelled here!
%            (IEEE Infocom 99) [Authors: Thomas Kuhn, Jan Jrjens]
% Domain   :
% Problem  :
% Version  :
% English  :

% Refs     :
% Source   :
% Names    :
% Status   : unknown
% Rating   : ?
% Syntax   : Number of formulae    :   26 (   6 unit)
%            Number of atoms       :   65 (  40 equality)
%            Maximal formula depth :   14 (   4 average)
%            Number of connectives :   40 (   1 ~  ;   0  |;  19  &)
%                                         (   0 <=>;  20 =>;   0 <=)
%                                         (   0 <~>;   0 ~|;   0 ~&)
%            Number of predicates  :    2 (   0 propositional; 1-2 arity)
%            Number of functors    :   16 (   8 constant; 0-2 arity)
%            Number of variables   :   60 (   0 singleton;  60 !;   0 ?)
%            Maximal term depth    :    6 (   1 average)

% Comments :
%          : tptp2X -f tptp -t add_equality tlsvariant.p
%------------------------------------------------------------------------

%----------------------- Asymmetrical Encryption ------------------------
input_formula(enc_equation,axiom,(
    ! [E1,E2] :
      ( (   knows(enc(E1, E2))
          & knows(inv(E2)) )
        => knows(E1) )  )).

%---------------------- Symmetrical Encryption ----------------------------
input_formula(symenc_equation,axiom,(
    ! [E1,E2] :
      ( (   knows(symenc(E1, E2))
          & knows(E2) )
        => knows(E1) )  )).

%--------------------------- Signature ------------------------------------
input_formula(sign_equation,axiom,(
    ! [E,K] :
      ( (   knows(sign(E, inv(K) ) )
          & knows(K) )
        => knows(E) )  )).

%---- Basic Relations on Knowledge where conc, enc, symenc and sign is included ----
input_formula(construct_message_1,axiom,(
    ! [E1,E2] :
      ( (   knows(E1)
          & knows(E2) )
        => (   knows(conc(E1, E2))
             & knows(enc(E1, E2))
             & knows(symenc(E1, E2))
```

```
            & knows(sign(E1, E2)) ) )  )).

input_formula(construct_message_2,axiom,(
    ! [E1,E2] :
      ( ( knows(conc(E1, E2)) )
        => (   knows(E1)
             & knows(E2) ) )  )).

%----------------------- Attackers Initial Knowledge -----------------------
input_formula(previous_knowledge,axiom,(
    (   knows(k_a)
      & knows(inv(k_a))
      & knows(k_ca)
    )   )).

%--------------------- TLS Main Protocol Specification --------------------
input_formula(tls_abstract_protocol,axiom,(
     ! [ArgS_11, ArgS_12, ArgS_13, ArgC_11, ArgC_12, DataC_KK, DataC_k, DataC_n, DataC_r] :
       (   % Client -> Attacker (1. message)
           (   knows(n)
             & knows(k_c)
             & knows(sign(conc(c, k_c), inv(k_c) ) )
           )
         & % Server -> Attacker (2. message)
           (    (   knows(ArgS_11)
                  & knows(ArgS_12)
                  & knows(ArgS_13)
                  & ( ? [X] : equal(sign(conc(X, ArgS_12), inv(ArgS_12) ), ArgS_13) )
                )
              => (   knows(enc(sign(conc(conc(kgen(ArgS_12), ArgS_11), ArgS_12), inv(k_s) ), ArgS_12 ) )
                   & knows(sign(conc(s, k_s), inv(k_ca) ) ) )
           )
         & % Cient -> Attacker (3. message)
           (    (   knows(ArgC_11)
                  & knows(ArgC_12)
                  & equal(sign(conc(s, DataC_KK), inv(k_ca)), ArgC_12 )
                  & equal(enc(sign(conc(conc(DataC_k, DataC_n), DataC_r), inv(DataC_KK) ), k_c), ArgC_11 )
                  & ( ? [DataC_ks] : equal(sign(conc(s, DataC_ks), inv(k_ca) ), ArgC_12 ) )
                  & equal(enc(sign(conc(conc(DataC_k, n), DataC_r), inv(DataC_KK) ), k_c), ArgC_11 )
                  & equal(enc(sign(conc(conc(DataC_k, DataC_n), k_c), inv(DataC_KK) ), k_c), ArgC_11 )
                )
              => (   knows(symenc(secret, DataC_k))
                 )
           )
)  )).
% ------------------------ Attack ----------------------------------------
% Here you can see that the secret can't be captured by an attacker!
%
input_formula(attack,conjecture,(
    knows(secret)  )).
```