Security Analysis of Crypto-based Java Programs using Automated Theorem Provers

Jan Jürjens*, Software & Systems Engineering, TU Munich

Abstract

Determining the security properties satisfied by software using cryptography is difficult: Security requirements such as secrecy, integrity and authenticity of data are notoriously hard to establish, especially in the context of cryptographic interactions. Nevertheless, little attention has been paid so far to the verification of such implementations with respect to the secure use of cryptography. We propose an approach to use automated theorem provers for first-order logic to formally verify crypto-based Java implementations, based on control flow graphs. It supports an abstract and modular security analysis by using assertions in the source code. Thus large software systems can be divided into small parts for which a formal security analysis can be performed more easily and the results composed. The assertions are validated against the program behavior in a run-time analysis. Our approach is supported by the tool JavaSec *available as open-source and validated in an application to a Java Card implementation of the Common Electronic Purse Specifications and the Java implementation Jessie of SSL.*

1 Introduction

Understanding the security goals provided by software making use of cryptography is one of the major challenges with security-critical systems. Since security requirements such as secrecy, integrity and authenticity of data are always relative to an unpredictable adversary, they are difficult to even define precisely, let alone to determine in a complex software system making sophisticated use of cryptographic operations for example to authenticate communication partners over an untrusted network. While a significant amount of research has been directed to develop formal techniques to analyze abstract specifications of crypto-based software (such as crypto protocols), few attempts have been made to apply the results developed in that setting to the analysis of implementations. Even if specifications exist for these implementations, and even if these had been analyzed formally, there is usually no guarantee that the implementation actually conforms to the specification.

To address this problem, we present an approach for analyzing crypto-based implementations for security requirements using automated theorem provers (ATPs) for first-order logic (FOL). Security requirements can not only be formalized straightforwardly in FOL, but verification is also powerful because of the efficient proof procedures of the ATPs. The Java code gives rise to a control flow graph in which the cryptographic operations are represented as abstract functions, and which is translated to formulas in FOL with equality. Together with a logical formalization of the security requirements, they are then given as input into any ATP supporting the TPTP input notation (such as e-SETHEO [SW00]). If the analysis reveals that there could be an attack, an attack generation script in Prolog is generated from the Java code. Our approach supports a modular security analysis by using assertions in the source code. Thus large software systems can be divided into small parts for which a formal security analysis can be performed more easily and the results composed. To validate that the assertions define a correct abstraction of these code fragments, we perform a run-time analysis. This way our approach can also be applied to code that calls libraries even if the code for the libraries is not (yet) available. Our approach is supported by the open-source tool JavaSec which automatically generates FOL formulas which are given as input to a variety of ATP's [jav]. The method has been validated at the industrial software project "Common Electronic Purse Specifications" implemented in Java Card [Jc] and the Java implementation Jessie of SSL [jes03].

Our goal is not to provide a full formal verification of Java code but to increase understanding of the security properties enforced by cryptoprotocol implementations in a way as automated as possible. Because of the abstractions, the approach may produce false alarms (which however have not surfaced yet in practical examples). Our focus here is on high-level security properties such as secrecy and authenticity, and not on low-level security flaws for example caused by buffer overflows (for which tools already exist).

*http://www4.in.tum.de/~juerjens – from October 2006: Open University, UK.

 DOI 10.1109/ASE.2006.60

2 Code Analysis

The analysis approach presented here works with the well-known Dolev-Yao adversary model for security analysis [DY83]. The idea is that an adversary can read messages sent over the network and collect them in his knowledge set. The adversary can merge and extract messages in the knowledge set and can delete or insert messages on the communication links. The security requirements can then be formalized using this adversary model. For example, a data value remains secret from the adversary if it never appears in the knowledge set of the adversary.

We explain the transformation from the Java program to FOL, which is given as input to the ATP. The corresponding tool-flow is shown in Fig. 1. Because of space limitations, we can not explain all steps in the transformation from the Java code to the logical formula in every technical detail. However, we will demonstrate them to the extent possible given the space at the hand of examples in Sect. 2.2 and 4.

We restrict our explanation to the analysis for secrecy of data. The idea here is to use a predicate knows which defines a bound on the knowledge an adversary may obtain by reading, deleting and inserting messages on vulnerable communication lines (such as the Internet) in interaction with the protocol participants. Precisely, $\mathsf{knows}(E)$ means that the adversary may get to know E during the execution of the protocol. For any data value s supposed to remain confidential, one thus has to check whether one can derive $\mathsf{knows}(s)$. From a logical point of view, this means that one considers a term algebra generated from data such as variables, keys, nonces and other data using symbolic operations including the ones in Fig. 2. There, the symbols E, E', and K denote terms inductively constructed in this way. These symbolic operations are the abstract versions of the cryptographic algorithms defined in the Java^{TM} Cryptography Architecture (JCA) [JCA]. Note that the cryptographic functions in the JCA are implemented as several methods, including

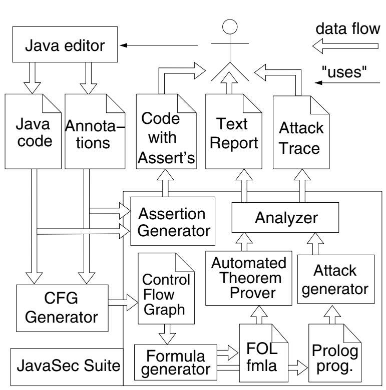

Figure 1. Tool-flow of the JavaSec **suite**

- $\mathsf{enc}(E, K)$ (encryption)
- $\mathsf{dec}(E, K)$ (decryption)
- $\mathsf{hash}(E)$ (hashing)
- $\mathsf{sign}(E, K)$ (signing)
- $\mathsf{ver}(E, K, E')$ (verification of signature)
- $\mathsf{kgen}(E)$ (key generation)
- $\mathsf{inv}(E)$ (inverse key)
- $\mathsf{conc}(E, E')$ (concatenation)
- $\mathsf{head}(E)$ and $\mathsf{tail}(E)$ (head and tail of concat.)

Figure 2. Abstract crypto operations

an object creation and possibly initialization. Relevant for our analysis are the actual cryptographic computations performed by the digest(), sign(), verify(), generatePublic(), generatePrivate(), nextBytes(), and doFinal() methods (together with the arguments that are given beforehand, possibly using the update() method), so the others are essentially abstracted away. Note also that the key and random generation methods generatePublic(), generatePrivate(), and nextBytes() are not part of the crypto term algebra in Fig. 2 but are formalized implicitly in the logical formula by introducing new constants representing the keys and random values (and making use of the $\mathsf{inv}(K)$ operation in the case of generateKeyPair()). In that term algebra, one defines the equations $\mathsf{dec}(\mathsf{enc}(\mathsf{E},\mathsf{K}),\mathsf{inv}(\mathsf{K})) = \mathsf{E}$ and $\mathsf{ver}(\mathsf{sign}(\mathsf{E},\mathsf{inv}(\mathsf{K})),\mathsf{K},\mathsf{E}) = \mathsf{true}$ for all terms E, K, and the usual laws regarding concatenation, head(), and tail() to hold. See [Jür04] for more information on this.

The predicates defined to hold for a given program are defined as follows. For each publicly known expression E, the statement knows(E) is derived. 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 cryptographic operations, formulas are generated for these operations for which some examples are given in Fig. 3. We use the TPTP notation for the FOL formulas, which is the input notation for many ATPs including the one we use (e-SETHEO [SW00]). Here & means logical conjunction and ![E1, E2] forall-quantification over E1, E2.

We explain how a Java program gives rise to a logical formula characterizing the interaction between the adversary and the protocol participants (the bottom left part of Fig. 1). We explain the translation first for a simplified fragment of Java without loops and concurrency. We then add

```
input_formula(construct_message_1,axiom,(
 ! [E1,E2]:
     ((knows(E1) & knows(E2))
       => (knows(conc(E1,E2)) & knows(enc(E1,E2))
          & knows(sign(E1,E2)))))).
input_formula(construct_message_2,axiom,(
 ! [E1,E2]:
(knows(conc(E1,E2)) => (knows(E1) & knows(E2))))).
```

Figure 3. Some general crypto axioms

these features in subsequent subsections below.

Also, to simplify the treatment of variables and their assignment, we first use standard transformations to simplify the translation from the program to logic. They are necessary, because in programming languages, program variables have state, while in classical logic variables are stateless.

side effects Side effects from method calls are flattened by traversing into the method definition. Where this becomes infeasible, one may add annotations to the method declaration (to be explained below) that abstractly capture the computation of a method (and its side effects).

static single assignment The program is transformed to the *static single assignment (SSA)* format as usual.

Below, setting a variable a to a value v will be formalized as the logical constraint $\mathsf{a} = \mathsf{v}$ on the models (which any valid model of the axioms will have to fulfill, whereby it amounts to an assignment). Getting the value from the variable a is modeled by just using that variable. We may ignore variable data definitions since they are not necessary in the TPTP input notation for the ATP. Similarly, we can treat variable initialization as assignment. In the case of local redefinitions of global variables, we assume a suitable renaming is used to avoid confusion.

To generate the FOL formula from the program, we first generate the control flow graph from the program. Note that the control flow graph is just a different representation of the program which easens further processing but still contains all relevant information. The logical formulas presented below could just as well be generated directly from the code (in particular we will use annotations below only to abstract away irrelevant parts to make the approach scale to realistic systems, not to generate the formulas).

We transform this graph to consist of graph transitions carrying labels of the form *await message e – check condition g – output message e'*. A graph transition is executed if a message conforming to its input pattern arrives (or if the input pattern is empty) and if its condition is satisfied. When the transition is executed, its action will be executed and then the next transition in the graph evaluated. In the label *await message e*, the expression *e* consists of a message name msg and a list of variables which will be assigned values when a message with name msg is received over the network. We use code annotations (defined in Sect. 2.1) to define which input variables store the incoming arguments of which messages, and which functions are used to receive them. Similarly, an *output message e'* pattern consists of a message name msg and a list of expressions, that are at run-time evaluated to values which are sent on the network as arguments of the message msg. Again we use code annotations defined below to specify which functions take care of sending out the messages. Lastly, we explain in Sect. 2.1 how to use other kinds of annotations to map an assignment assgmt of an expression to a variable in Java to a logical predicate $\mathsf{p}_{\mathsf{assgmt}}$ on the corresponding logical variable. The list of arguments of the message *e* may be empty and condition *g* equal to true where they are not needed. See Sect. 4 for examples how this is done in practical applications.

$$
\begin{aligned}
&\mathsf{PRED}(l) = \\
&\forall exp_1, \ldots, exp_n.\ \Big(\mathsf{knows}(exp_1) \wedge \ldots \wedge \mathsf{knows}(exp_n) \\
&\qquad \wedge\ cond(exp_1, \ldots, exp_n) \\
&\qquad \Rightarrow \mathsf{knows}(exp(exp_1, \ldots, exp_n) \\
&\qquad\quad \wedge\ \mathsf{PRED}(l')\Big)
\end{aligned}
$$

Figure 4. Transition predicate

For the mapping from the control graph defined above to a FOL formula, we map the Boolean expression in Java syntax to logical syntax in the TPTP format, e.g. by replacing the equality test == to the binary Boolean function equal() and similarly for the other Boolean connectives.

Suppose now that we are given a graph transition $l = (\mathsf{source}(l), \mathsf{guard}(l), \mathsf{msg}(l), \mathsf{target}(l))$ 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. Suppose that the transition l' is the next transition in the graph. For each such transition l, we define a predicate $\mathsf{PRED}(l)$ as in Fig. 4. If a next transition l' does not exist, $\mathsf{PRED}(l)$ is defined by substituting $\mathsf{PRED}(l')$ with true in Fig. 4.

The formula formalizes 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 protocol participants to receive the message $exp(exp_1, \ldots, exp_n)$ in exchange, and then the protocol continues. 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. To construct the recursive definition above, we assume that the control flow graph is finite and cycle-free. The case of loops is explained below.

For each object O in the system to be analyzed, this gives a predicate $\mathsf{PRED}(O) = \mathsf{PRED}(l)$ where l is the first transition in the control flow graph of O. The axioms in the overall FOL formula for a given protocol are then the conjunction of the formulas representing the publicly known expressions, the formula in Fig. 3, and the conjunction of the formulas $\mathsf{PRED}(O)$ for each object O in the protocol.

The formulas defined above are written into the TPTP file as axioms. The security requirement to be checked is written into the TPTP file as a conjecture (for example, knows(secret) in case the secrecy of the value secret is to be checked). The ATP will then check whether the conjecture is derivable from the axioms. In the case of secrecy, the result is interpreted as follows: If knows(secret) can be derived from the axioms, this means that the adversary may potentially get to know secret. If the ATP returns that it is not possible to derive knows(secret) from the axioms, this means that the adversary will not get secret (relative to our system (and in particular adversary) model – a 100% proof of the security of a system in reality is neither the goal of this approach, nor in general achievable).

SETHEO is an efficient ATP for FOL in clausal normal form based on the model eliminiation calculus and extended to e-SETHEO for reasoning about equality properties (e.g. making use of lazy basic paramodulation and linear completion). We use e-SETHEO for verifying security protocols as a "black box" and only make use of the standard proof techniques that it implements: 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 e-SETHEO interchangingly with any other ATP accepting TPTP as an input format (such as SPASS, Vampire and Waldmeister) when it may seem fit. More information on the verification of the security properties can be found in [Jür05] where this approach was applied to UMLsec specification models, and a comparison with other approaches in Sect. 5.

Loops and Recursion In general, the control flow graph considered above may contain loops. These may arise from the relevant commands in the code such as while or for, or backward goto jumps, or through recursion flattened in the control flow graph. In software verification, loops are often only investigated through a bounded number of rounds (which is a classical approach in automated software verification, see for example [HS01]). Since in general there may be unbounded loops in the Java program, this can only be achieved in an approximate way by fixing a natural number n (supplied by the user of the approach) and unfolding all cycles up to the transition path length n. The analysis process can also be iterated with n as the iteration variable to approximate the unbounded loops as far as possible (within the limits of tool performance).

We go beyond that in the context of our security analysis approach by making use of a refinement of the static single assignment format going back for example to the idea of "history variables" in [Cli73]. Originating in Hoare logic style verification, it seems to have attracted limited attention so far in FOL based verification. The idea, intuitively, is to replace the variables in loops after the translation to logic by infinite arrays indexed by a loop counter (or more precisely by functions from natural numbers to the set of array variable values). Here we restrict our attention to loops which are well-structured in the sense that they have an entry point where the iteration counter may be introduced and incremented (for example, for or while loops). For example, Fig. 5 shows a simple fragment of a Java method containing an unbounded loop with some assignments. This is first translated to the SSA format, and then to the logical formula in the TPTP format. In case of nested loops, one needs to use multi-dimensional arrays.

```
Example:                     SSA:
while (true)                 while (true)
{ k = a + 1;                 { k = a0 + 1;
  a = b + k;                   a1 = b0 + k;
  b = b + 1; }                 b1 = b0 + 1; }

TPTP:
input_formula(ForLoop_axiom_ID1,axiom,(
![I]: (equal (k[I], sum(a0[I],1)) &
       equal (a1[I], sum(b0[I],k[I])) &
       equal (b1[I], sum(b0[I],1)) &
       equal (a0[succ(I)],a1[I]) &
       equal (b0[succ(I)],b1[I])))).
```

Figure 5. Loop example

Note that we do not have to worry about manually finding loop invariants, since we use ATPs. Although of course loops are in general undecidable to verify, this problem has not become apparent in our applications yet, because for crypto protocols, on our level of abstraction, the emphasis is on interaction rather than computation. The treatment of recursion works in a similar way; we have to omit the details here because of space limits.

Concurrent threads For concurrent threads, we identify maximal transition paths in the control flow graph between synchronization points (where shared variables are written or read). We consider all possible interleavings between the maximal transition paths by constructing a formula ϕ consisting of nested implications as in Fig. 4 but containing predicates PRED(Pi) where i ranges from 1 to the number of paths n. We consider the all-quantification of the formula $\psi \Rightarrow \phi$ over the possible interleavings of the paths (represented as ordered lists of the numbers 1 through n), where ψ is an equational formula assigning to the predicate PRED(Pi) the values from the predicate formalising the path numbered j, where j is the ith element in the ordered list. The predicates PRED(TR) for all such transitions TR are then joined together using logical conjunctions. The resulting logical formula is closed by forall-quantification over all free variables contained.

Limitations Since the adversary knowledge set is approximated from above (because one abstracts away for example from the message sender and receiver identities), one will find all possible attacks observable in our system model, but one may also encounter "false alarms". However, this has not so far happened with practical examples, and the 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, but instead the ATP may not return a result at all.

In our practical applications of our method, this limitation has, however, not yet become observable.

Attack Generation In case the result is that there may be an attack, in order to fix the flaw in the code, it would be helpful to retrieve the attack trace. Since ATPs such as e-SETHEO are highly optimized for performance by using abstract derivations, it is not trivial to extract this information. Therefore, we also implemented a tool which transforms the logical formulas explained above to Prolog (the bottom right part of Fig. 1). While the analysis in Prolog is not useful to establish whether there is an attack in the first place (because it is in order of magnitudes slower than using e-SETHEO and in general there are termination problems with its depth-first search algorithm), Prolog works fine in the case where one already knows that there is an attack, and it only needs to be shown explicitly (because it explicitly assigns values to variables during its search, which can then be queried).

2.1 Annotations

We now explain how to use code annotations for several purposes:

- to logically and abstractly formalize standard Java constructs important to our analysis, such as cryptographic algorithms and communication functions,
- to provide the user of our approach with an extension mechanism that supports abstraction and modularity in order to make (as far as possible automated) formal verification feasible for industrial-size software,
- to provide a way to proceed in case certain functions used in the program are not available as source code,
- to enable a security analysis during development when only part of the code has been constructed so far and correction is still less costly.

The annotations are defined as Java comments and can thus be included into the Java source code (since the generation of control flow graphs keeps the comments). Each annotation is supposed to specify how a Java method or variable should be abstracted when the FOL formula is generated from the Java code.

An annotation, whose syntax is given in Fig. 6, starts with the key word //@J2SD_ANN followed by the name of the method or variable. Then the keyword //@J2SD_CONN follows (optionally) which specifies the trigger, the guard, and the effect of a transition which should be inserted in the control flow graph where the method is called or variable is used. The trigger of a transition specifies the message that has to be received so that the transition is executed, provided the guard holds. The effect gives the message that will be sent out when the transition is executed. One can refer to the arguments of the method which

```
//@J2SD_ANN (<<method name>>)
//@J2SD_CONN (<<trigger>>; <<guard>>; <<effect>>)
//@J2SD_INSERT (<<value>>)
//@J2SD_AXIOMS
// <<FOL axioms>>
//@J2SD_AXIOMS_END
```

Figure 6. Code annotations

appear at the occurence which should be replaced by identifying them as meth_i where meth is the name of the method as specified with the key word //@J2SD_ANN and i the number of the argument. The keyword //@J2SD_INSERT specifies an expression that should be inserted at the place of the method call as its return value, or in place of the variable, respectively. The definition ends with the optional keyword //@J2SD_AXIOMS which allows one to insert FOL formulas axiomatizing the expressions used in the graph transition and the inserted value.

An example for an annotation for the method RecvMsg is given in the top of Fig. 7. When applied at the occurrence Success = Receiver.RecvMsg(Buff) of this method, this leads to the control flow graph transition annotated with the trigger recv(RecvMsg), followed by the transition which returns the return value true. An example for an annotation defining a variable is given in the bottom of Fig. 7. Here the Java constant g_Cert storing a cryptographic certificate should be replaced by its abstract definition sign(conc(c, k_c), inv(k_ca)), specifying that it should be the signature of the concatenation of the identity c and the corresponding public key k_c using the private key inv(k_ca) of a certification authority. For example, when the constant is used in the Java statement Success = Client.SendMsg(g_Cert); the resulting graph transition is annotated by the guard Success = Client.SendMsg(sign(conc(c, k_c), inv(k_ca))) (assuming there is no annotation specifying that e.g. Client.SendMsg be replaced as well).

There are standard method annotations which map methods in the standard libraries to their representations in the state machine. For example, the operator == is mapped to the logical = operator. The treatment of assignment was already explained below, as well as the substitution from the cryptographic functions in the JCA by their abstract counter-parts in Fig. 2, using axioms some of which are given in Fig. 3. Similarly, the arithmetic operators are as far as possible abstracted away, because of their challenges to automated verification (arithmetic is in general

```
//@J2SD_ANN (RecvMsg)
//@J2SD_CONN (recv(RecvMsg); ;)
//@J2SD_INSERT (true)

//@J2SD_ANN (g_Cert)
//@J2SD_INSERT (sign(conc(c, k_c), inv(k_ca)))
```

Figure 7. Method and variable annotations

undecidable). Since for a security analysis of cryptoprotocols on our level of abstraction, the emphasis is on interaction rather than computation of number theoretic algorithms, this is not a problem in practice, as shown with our evaluations in Sect. 4. Where possible, we make use of the more tractable Presburger arithmetic which is currently built into many ATPs such as e-SETHEO.

Validating assertions To validate that the assertions are included correctly into the source code, we use a run-time analysis approach: Using a precompiler built for this purpose, the assertions are compiled into run-time checks implemented with the Java command assert that throw an exception if violated during the test-runs of the software (the top middle part of Fig. 1). This is done by transforming the logical conditions to the corresponding expressions in the Java syntax. Also, the abstract crypto representations are mapped to their concrete implementations in the JCA.

By extensive testing, one can then ensure that the conditions are not violated during execution of the software, which ensures that they are sound with respect to the detailed source code. Consequently, it means that the formal analysis performed based on the assertions is also sound with respect to the detailed source code.

Since testing can usually not be exhaustive, one may also leave in the run-time checks during actual deployment of the software (which leads to a performance penalty, which however to our experience is quite bounded). Alternatively, one may use an interactive theorem prover (such as Isabelle) to exhaustively prove that the assertions are sound with respect to a formal semantics of the Java language. In ongoing work, we are currently aiming to automate this by again making use of ATPs for FOL.

2.2 Demonstration at TLS variant

We consider a variant of the handshake part of the Internet security protocol TLS proposed in [APS99]. TLS is the current version of the Internet security protocol SSL. A simple Java implementation of the client side of a variant of the TLS protocol from [APS99] is in Fig. 8. In this abstracted version, the cryptographic operations from the JCA are already substituted by the abstract operations in Fig. 2 using the annotations some of which were shown in Fig. 7.

The protocol aims to establish a secure channel over an untrusted communication link between a client and a server. This channel is supposed to provide secrecy and server authenticity. Both client and server can run the protocol with arbitrary servers and clients. The threat scenario here is that the adversary controls the communication link between client and server. In our analysis, this is captured by enabling the adversary to read, delete, and insert messages at the corresponding communication link.

```
public static void TLS_Client(String secret) {
// C->S: Init
send(n);
send(k_c);
send(sign(conc(c,k_c),inv(k_ca)));
// S->C: Receive Server's respond
recv(Resp_1);
recv(Resp_2);
// Check Guards
if (equal(fst(ext(Resp_2,k_ca)),s) &&
    equal(snd(ext(dec(Resp_1,inv(k_c)),
                    snd(ext(Resp_2, k_ca)))),n))
  { // C->S: Send Secret
     send(enc(secret,fst(ext(dec(Resp_1,inv(k_c)),
                    snd(ext(Resp_2,k_ca)))))); }}
```

Figure 8. Abstracted client code

Using the annotations, the program is transformed to the FOL formulas as explained above. An excerpt from the resulting TPTP file is given in Fig. 9. The protocol itself is expressed by a for-all quantification over the pieces of messages which are transferred over the communication channel. The message variables Init_1, Init_2, Init_3, Resp_1, Resp_2, Xchd_1 stand for the values received as arguments of the messages Init, Resp, Xchd, respectively. Each message exchange (the first and third one sent from the client, the second one from the server) is represented by an implication.

When given the formulas generated from the source code together with the conjecture knows(secret), the prover returns Proof found (within a second by the eprover contained in the e-SETHEO suite, see Fig. 11), which means that the secrecy requirement on the value secret is not fulfilled. We should stress again that this does not concern the main-stream TLS implementation, but the variant proposed in [APS99]. The message flow diagram corresponding to this man-in-the-middle attack, which is generated by the Prolog-based attacker generator, is given in Fig. 10.

We propose to change the protocol by substituting $k :: N$ in the second message by $k :: N :: K_C$ and by including a check regarding this new message part at the client. When one repeats the security analysis process, it turns out that the repaired protocol now provides secrecy of the transmitted secret (with respect to our adversary model).

```
E-SETHEO csp04 single processor running
(c) 2004 Technische Universitat Munchen
...
executing .../bin/eprover
analyzing results ...
proof found
status SUCCESS consuming 1 seconds
e-SETHEO done. exiting
```

Figure 11. Verification result for TLS variant

```
input_formula(protocol,axiom,(
![Init_1,Init_2,Init_3,Resp_1,Resp_2,Xchd_1]:
( % C -> Attacker
  (knows(n) & knows(k_c) & knows(sign(conc(c,k_c),inv(k_c)))
    & (knows(Resp_1) & knows(Resp_2) & equal(fst(ext(Resp_2,k_ca)),s)
         & equal(snd(ext(dec(Resp_1,inv(k_c)), snd(ext(Resp_2, k_ca)))),n)
       => knows(enc(secret,fst(ext(dec(Resp_1,inv(k_c)),snd(ext(Resp_2,k_ca))))))))
  & % S -> Attacker
    (((knows(Init_1) & knows(Init_2) & knows(Init_3) & equal(snd(ext(Init_3,Init_2)),Init_2))
       => knows(conc(enc(sign(conc(kgen(Init_2),Init_1),inv(k_s)),Init_2),sign(conc(s,k_s),inv(k_ca))))
          & knows(Xchd_1)))).
```

Figure 9. Core protocol axiom

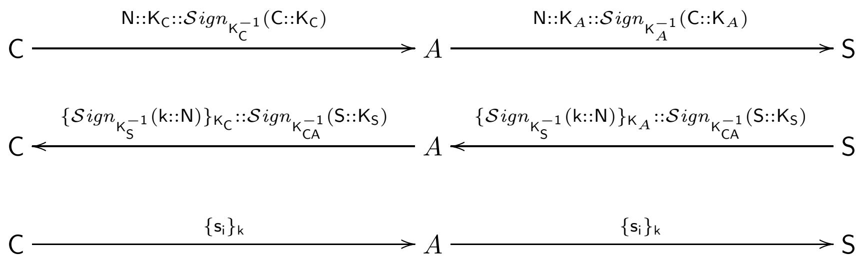

Figure 10. Attack message flow

3 Modular Verification using Assertions

We explain how one can perform a modular security analysis with our approach by including security assertions in the program parts to be composed which are generated during the security analysis of these parts. Note that we are currently focussing on compositional verification for Dolev-Yao type properties of crypto-based software; a generalization to other properties or kinds of software would be very interesting but is beyond the scope of the current work. Again, here we have to restrict ourselves to confidentiality of data for space reasons. Note also that our goal is not to generally solve the problem of compositionality of security properties, which is well known to be a very difficult problem (see e.g. [Jür00] for some work in that direction). More modestly, we simply aim to collect the logical formulas generated from different program fragments together and store them in assertions to be reused in a later analysis. Our aim is not to prove that a security requirement such as confidentiality is compositional in general.

A set of security assertions for a program part p consists of statements $\mathsf{derived}(\mathsf{L},\mathsf{C},\mathsf{E})$ where L is a list of variables, C is a condition over the variables in L, and E is an expression which may contain free variables from L.

These assertions mean that the set of adversary knowledge is contained in the set of expressions E that can be constructed by instantiating the variables from L with values that themselves can be derived this way for p and which fulfill the condition C. This way of formulating modular assertions on parts of the code is inspired by the Assumption/Commitment specification approaches (e.g. [BS01]).

More formally, a program fragment p has an associated set $\mathcal{L}$ of statements $\mathsf{derived}(\mathsf{L},\mathsf{C},\mathsf{E})$ if according to the security analysis in Sect. 2, an adversary gets to know only those expressions that can be constructed recursively in the following way.

- For all valuations of the variables v in L fulfilling C and such that $\mathsf{knows}(\mathsf{v})$ holds, we have $\mathsf{knows}(\bar{\mathsf{E}})$ for the corresponding valuation $\bar{\mathsf{E}}$ of E.

Note that for a single protocol run of p, it is sufficient to use a finite set of such assertions, namely those defined in the following way:

- For each expression E in the initial knowledge set of the adversary or the set of previously known expressions for p, we define an assertion of the form $\mathsf{derived}([\,],\mathsf{true},\mathsf{E})$ where $[\,]$ is the empty list and true is the Boolean constant.
- For each list of predicates of the form

  $$\mathsf{PRED}(\mathsf{TR}_\mathsf{k}) \equiv \bar{\mathsf{i}}_\mathsf{k} \wedge \mathsf{c}_\mathsf{k} \Rightarrow \bar{\mathsf{a}}_\mathsf{k} \wedge \mathsf{PRED}(\mathsf{TR}_{\mathsf{k}+1})$$

  defined for p in Sect. 2 for k up to some number n, we define an assertion of the form $\mathsf{derived}(\mathsf{L},\mathsf{C},\mathsf{E})$ where L is the list of free variables in that list of predicates, C is a nested implication of the form

  $$\mathsf{IMP}(\mathsf{TR}_\mathsf{k}) \equiv \mathsf{c}_\mathsf{k} \Rightarrow \bar{\mathsf{a}}_\mathsf{k} \wedge \mathsf{IMP}(\mathsf{TR}_{\mathsf{k}+1})$$

  (where $\bar{\mathsf{a}}_\mathsf{k} = \mathsf{a}_\mathsf{k}$ in case a_k is of the form $\mathsf{localvar} = \mathsf{value}$ and $\bar{\mathsf{a}}_\mathsf{k} = \mathsf{true}$ otherwise), and E is the concatenation of the actions a_k that are of the form $\mathsf{ak} = \mathsf{outpattern}$.

For atoms that are freshly generated in each protocol run, we assume that these are given as methods with a sequence number of the protocol run as free variables. Then one can obtain a finite set of assertions bounding the adversary knowledge by closing the above assertions with forall quantification over the sequence number variables that they contain.

In order to make sure that the set of expressions generated by $\mathcal{L}$ does not contain any expressions that according to the security requirements for p are supposed to remain secret (because otherwise the security assertions would be practically unusable), the code fragment p is first analyzed using the approach in Sect. 2. We then say the $\mathcal{L}$ is a *secure bound generator for the adversary knowledge*. Of course, there can only be a secure bound generator for a program fragment p if p in fact fulfills its secrecy requirements.

To analyze a program fragment p carrying a set of assertions $\mathcal{L}$ one takes the formulas generated from the approach in Sect. 2 and adds for each assertion of the form derived(L, C, E) an axiom of the form

```
![v1,...,vn]: knows(v1) &...& knows(vn)
              & C(v1,...,vn) => knows(E)
```

to the TPTP file that is to be analyzed by the ATP (where L is assumed to be the list of variables v1, ..., vn and C(v1, ..., vn) is the instantiation of the condition C with the variables v1, ..., vn).

4 Evaluation in Case Studies

4.1 Common Electronic Purse Specific.

We applied our method and tool to a security analysis of the Common Electronic Purse Specifications (CEPS) [CEP01], a candidate for a globally interoperable electronic purse standard supported by organizations representing 90 % of the world's electronic purse cards (including Visa International). Stored value smart cards, called "electronic purses", allow cash-free point-of-sale (POS) transactions with transaction-bound authentication performed by the built-in chips and where the account balance is stored and updated on the card. The application software is implemented in Java, the card software in Java Card, for which our approach works with the necessary adjustments. Here we consider the purchase transaction, an off-line protocol which allows the cardholder to use the electronic value on a card to pay for goods.

We considered a prototypical implementation of a part of the CEPS specification consisting of about 600 KB of source code. The class MsgChannel implements a communication channel between a sender and a receiver. Using the method send(inmsg : Message) : Message (where the class Message defines the messages), the sender can submit a message to the channel, which is sent to the receiver. The reply the the message is given back to the sender as the return value of the send() method. As explained in previous sections, we use annotations to abstract away the actual definition of the send() method and substitute it with the abstract behavior as relevant to our analysis (namely that a message a sent out and the reply is received).

```
E-SETHEO csp04 single processor running
(c) 2004 Technische Universitat Munchen
...
executing .../bin/dctp
analyzing results ...
model found
status SUCCESS consuming 1 seconds
e-SETHEO done. exiting
```

Figure 12. Verification result for CEPS

According to the specification, a relevant threat scenario is that the attacker is able to access the POS device links, and can access other Purchase Security Application Modules (PSAMs) over the Internet, but is not able to tamper with the smart cards. We performed the security analysis regarding this threat scenario, with respect to the security requirement that (informally speaking) the merchant will not loose any money during a transaction. This kind of security requirement is formalized directly in the conjecture in the TPTP file (contrary to the case of secrecy explained above where the conjecture represents the *in*security of the system). We performed the security analysis on the control flow graph after the abstractions defined by the annotations. The output of e-SETHEO given in Fig. 12 shows that the ATP dctp contained in the e-SETHEO suite was able to show within 1 second computation time that the conjecture cannot be derived from the axioms ("model found" means that there is a counter-example found that satisfies the axioms but not the conjecture). With respect to this formalization, this means that the security requirement is in fact violated. As we could see from the attack trace generated from the Prolog-based attack generator, the core of the attack is that the attacker redirects the messages between the card C and the PSAM P to another PSAM P′ (for example with the goal of buying electronic content) and to let the cardholder pay for it. The attack has a good chance of going undetected: the cardholder will not notice anything suspicious, because the deducted amount is correct. Also, the identifiers registered by the card are non-self-explanatory data that the cardholder cannot be assumed to be able to verify, and the card itself has no information about what the correct identity should be. The merchant who owns P will notice only later a lacking amount of money. The CEPS security working group has been informed about the problem. More detailed results can be found at [jav].

4.2 Jessie: An Open-Source SSL Project

We also applied our method to the open implementation of the Internet security protocol SSL in the project Jessie, which is a free implementation of the Java Secure Sockets Extension, the JSSE. SSL is the de facto standard for securing http connections, which however has been the source of several significant security vulnerabilities in the past and is therefore an interesting target. The whole Jessie project currently consists of about 5 MB of code, but the part directly relevant to SSL consists of less than 700 KB in about 70 classes. Therefore it is challenging, but manageable for formal analysis.

Setting up the connection is done by two methods: doClientHandshake() on the client side and doServerHandshake() on the server side, which are part of the SSLsocket class in jessie − 1.0.1/org/metastatic/jessie/provider. After some initializations and parameter checking, both methods perform the following interaction between client and server:

```
ClientHello            -->
ServerHello           <--
Certificate*          <--
ServerKeyExchange*    <--
CertificateRequest*   <--
ServerHelloDone*      <--
Certificate*           -->
ClientKeyExchange      -->
CertificateVerify*     -->
[ChangeCipherSpec]     -->
Finished               -->
[ChangeCipherSpec]    <--
Finished              <--
```

Each of the messages is implemented by a class, whose main methods are called by the doClientHandshake() resp. doServerHandshake() methods. The above messages are thus mapped directly to a control flow graph which can be analyzed using our approach.

```
//@J2SD_ANN (ClientHello.write)
//@J2SD_CONN (;; send(SSL3_MT_CLIENT_HELLO))
//@J2SD_CONN (;; send(msg_len_3bytes))
//@J2SD_CONN (;; send(version_high,version_low))
//@J2SD_CONN (;; send(new_randon))
//@J2SD_CONN (;; send(session_id_len))
//@J2SD_CONN (;; send(session_id))
//@J2SD_CONN (;; send(supported_ciphers))
//@J2SD_CONN (;; send(supported_compression+1))
//@J2SD_CONN (;; send(compression_ids))
//@J2SD_CONN (;; send(0))
//@J2SD_INSERT 1
```

Figure 13. Annotation for ClientHello.write

```
E-SETHEO csp04 single processor running
(c) 2004 Technische Universitat Munchen
...
executing .../bin/eprover
analyzing results ...
model found/total failure
status SUCCESS consuming 2 seconds
e-SETHEO done. exiting
```

Figure 14. Verification result for Jessie

Communication is implemented as follows: With the method call msg.write(dout, version), the message msg is written into the output buffer dout. Using a suitable annotation, each occurrence of such a method call is substituted by the abstract function send(msg) in the control flow graph generated from the source code. The method call dout.flush later flushes the buffer. The assignment msg = Handshake.read reads a message from the buffer during the handshake part of the protocol. As another example for an annotation, the one for the method ClientHello.write can be found in Fig. 13.

We verified the abstracted control flow graph against the relevant security requirements such as secrecy and authenticity using our tools. In each case, the properties were proved within less than a minute. For example, the verification of the secrecy of the master secret communicated in the SSL protocol took 2 seconds and was achieved by the eprover contained in the e-SETHEO suite (see Fig. 14). Here "model found" means that there exists a counter-example to the conjecture (that the adversary can get to know the master-secret) given the axioms. "total failure" means that this was established by exhaustively trying all possibilities to derive the conjecture from the axioms. More detailed results can be found at [jav].

5 Related Work

There has been a significant amount of work on applying formal security analysis on the specification level, some of it making use of FOL. [Sch97] formalizes the well-known BAN logic in FOL and uses the ATP SETHEO to proof statements in the BAN logic. This is different from our approach which is based on the knowledge of the adversary, instead of the beliefs of the protocol participants. [Wei99] analyzes the Neuman-Stubblebine key exchange protocol using first-order monadic Horn formulas and the ATP Spass. This approach differs from ours for example in that in general we also admit non-monadic Horn formulas, to be able to consider unbounded state when necessary to express a security property. [Coh03] uses first-order invariants to verify cryptographic protocols against safety properties. The approach is supported by the ATP TAPS. Compared to our approach, the method does not generate counter-examples (that is, attacks) in case a protocol is found to be insecure. Other, less directly related approaches include [Bla01] (based Prolog) and [Aea05] (using a SAT solver). The approach used here was introduced for the case of specification-level analysis and compared to existing approaches in [Jür05]; some early ideas in the context of C-based crypto software have been sketched in [JY05]. It would be interesting to investigate how the other approaches could also be applied to source code verification in the way proposed here.

Previous work on programming-language based security has mostly focussed on security properties such as secure information flow (rather than verification of of cryptographic protocol implementations), e.g. [Mye99, HVY00]. There has been a lot of work on providing tool-support for verifying Java programs related to the Java Modeling Language (JML), see [BCC+05]. To the extent of our knowledge, JML has not been used for cryptographic protocols yet. It would be very interesting to try to integrate our approach into the JML setting.

6 Conclusion

We use automated theorem provers for first order logic to understand the security requirements provided by Java implementations of cryptographic protocols. Our approach constructs a logical abstraction of the annotated code which can be used to analyze the code for security properties (such as confidentiality) with ATPs. It supports a modular security analysis of crypto-protocol implementations using assertions in the source code.

By presenting experiences from two industrial-size case-studies in Sect. 4, we argue that the proposed techniques are not overly complex and sufficiently general to be applicable in practice. Although our approach is not completely automatic and requires some effort for annotating the code to make it scale, it turned out to be applicable with reasonable effort even in relatively large software projects, as demonstrated at the hand of a prototypical implementation of the Common Eletronic Purse Specifications and the open-source implementation Jessie of the SSL protocol in Sect. 4. We keep the annotation effort bounded by providing an annotated standard library.

The main restriction of the approach is probably the necessity to understand parts of the source code to be able to include annotations. In ongoing work we are therefore investigating how to generate annotations from UMLsec specifications [Jür04] and how to automatically verify the code against these annotations by making use of run-time verification, testing, and automated local formal verification. This should reduce the burden of the level of program understanding currently required.

Acknowledgements Helpful discussions with Manfred Broy, Daniel Ratiu, and Gernot Stenz are gratefully acknowledged, as well as help from David Kirscheneder, Axel Heider, Jun Yuan, Juan Wang, and Sebastian Frühling with the examples in this paper, and comments by the anonymous reviewers which lead to significant improvements in the presentation of the paper.

References

[Aea05] A. Armando and D.A. Basin et al. The AVISPA tool for the automated validation of internet security protocols and applications. In *CAV 2005*, volume 3576 of *LNCS*, pages 281–285. Springer, 2005.

[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, March 1999.

[BCC+05] L. Burdy, Y. Cheon, D.R. Cok, M.D. Ernst, J.R. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, 2005.

[Bla01] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001)*, pages 82–96. IEEE, 2001.

[BS01] M. Broy and K. Stølen. *Specification and Development of Interactive Systems*. Springer, 2001.

[CEP01] CEPSCO. Common Electronic Purse Specifications, 2001. Business Requirements Version 7.0, Functional Requirements Version 6.3, Technical Specification Version 2.3, available from http://www.cepsco.com.

[Cli73] M. Clint. Program proving: Coroutines. *Acta Informatica*, 2:53–60, 1973.

[Coh03] E. Cohen. First-order verification of cryptographic protocols. *Journal of Computer Security*, 11(2):189–216, 2003.

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

[HS01] G.J. Holzmann and M.H. Smith. Software model checking: extracting verification models from source code. *Software Testing, Verification & Reliability*, 11(2):65–79, 2001.

[HVY00] K. Honda, V.T. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In G. Smolka, editor, *ESOP 2000*, volume 1782 of *LNCS*, pages 180–199. Springer, 2000.

[jav] JavaSec tool. http://www4.in.tum.de/˜javasec.

[Jc] http://java.sun.com/products/javacard/index.jsp.

[JCA] Java^{TM} cryptography architecture. http://java.sun.com/j2se/1.5.0/docs/guide/security/CryptoSpec.html.

[jes03] Jessie: A free implementation of the JSSE, 2003. http://www.nongnu.org/jessie.

[Jür00] J. Jürjens. Secure information flow for concurrent processes. In *CONCUR 2000*, volume 1877 of *LNCS*, pages 395–409. Springer, 2000.

[Jür04] J. Jürjens. *Secure Systems Development with UML.* Springer, 2004.

[Jür05] J. Jürjens. Sound methods and effective tools for model-based security engineering with UML. In *ICSE 2005*. IEEE, 2005.

[JY05] J. Jürjens and M. Yampolskiy. Code security analysis with assertions. In *ASE 2005*, pages 392–395. ACM, 2005.

[Mye99] A. Myers. Jflow: Practical mostly-static information flow control. In *26th ACM Symposium on Principles of Programming Languages (POPL)*, 1999.

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

[SW00] G. Stenz and A. Wolf. E-SETHEO: An automated^3 theorem prover. In *TABLEAUX 2000*, volume 1847 of *LNCS*, pages 436–440. Springer, 2000.

[Wei99] C. Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In *CADE-16*, volume 1632 of *LNCS*, pages 314–328, 1999.