Using Interface Specifications for Verifying Crypto-protocol Implementations

Jan Jürjens*

Computing Department, Open University, GB

Abstract. An important missing link in the construction of secure systems is finding a practical way to establish a correspondence between a software specification and its implementation. In this paper, we show to make use of interface specifications to address this problem for the case of crypto-based Java implementations (such as crypto protocols). We explain this using an approach using automated theorem provers for first-order logic which links the implementation to a specification model, which has been presented in earlier work. We present the details at the hand of an application to the open-source Java implementation Jessie of the SSL protocol. In ongoing work, we apply the approach to the standard Java Secure Sockets Extension (JSSE) library that was recently open-sourced by Sun.

1 Introduction

Many security incidents at the software level have been reported, sometimes with potentially quite severe consequences. Any support to aid secure systems development is thus dearly needed. With respect to crypto-based software (such as crypto protocols or software using cryptographic signatures), a lot of very successful work has been done to formally analyze abstract specifications of these protocols for security design weaknesses, including [Low96, LR97, AG99, Pau98, BBD+05] (cf. [RSG+01, Jür04] for overviews). More recently, there has been some work towards verifying implementations of security-critical software in general [HALM06], and also in particular of crypto protocols [JY05, GLP05], [Jür06, BFGT06]. While the approaches reported in [GLP05, BFGT06, Gor06] aim to verify implementations written by the research groups themselves (and [HALM06] does not address crypto protocols), the line of work reported in the current paper (which was started in [JY05, Jür06, Jür07]) is targeted to legacy implementations of crypto protocols. It is motivated by the fact that so far, crypto-based software is usually not generated automatically from formal specifications, or even created in ways under control of the security analyst. Thus, even where the corresponding specifications are formally verified, the implementations may still contain vulnerabilities related to the insecure use of cryptographic algorithms. An example for a crypto protocol whose design had been

* `http://www.jurjens.de/jan` . This work was partially funded by the Royal Society within the project Modelbased Formal Security Analysis of Crypto Protocol Implementations.

formally verified for security and whose implementation was later found to contain a weakness with respect to its use of cryptographic algorithms can be found in [RS98].

The current paper uses an approach for analyzing crypto-based implementations for security requirements using automated theorem provers (ATPs) for first-order logic (FOL), which is based on earlier work reported in [Jür05]. Security requirements can be formalized straightforwardly in FOL, and the ATPs offer efficient derivation algorithms. The Java code is linked to a specification model 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 (which is a standard for formulating FOL formulas for ATPs), such as e-SETHEO [SW00] or SPASS [WBH+02]. The approach supports a modular security analysis by using assertions in the source code. Where a verification fails because the implementation contains a security flaw, one can use a Prolog engine to generate the corresponding attack trace.

Our goal is not to provide a full formal verification of Java code but to increase understanding of the security properties enforced by crypto protocol implementations in a way which is as automated as possible. For the moment, we assume that the cryptographic algorithms called in the crypto protocol implementations have been implemented correctly (and our goal is to verify that they are used correctly in the crypto protocol). Also, because of the abstractions introduced for efficiency reasons (as explained below, for example abstracting from the identity of the send of a message), the approach may produce false alarms (which however have not surfaced yet in practical examples).

The goal of the current paper is to explain how this line of work fits in with the increasing body of work on verification using rich interface specifications. Throughout, we use an application of the approach to the open-source Java implementation Jessie of the SSL protocol as a running example. In ongoing work, we apply the approach to the standard Java Secure Sockets Extension (JSSE) library that was recently open-sourced by Sun.

2 Verification using Interface Specifications

The goal in using rich interface specifications (sometimes also called behavioural interface specifications) is to bridge the gap between the efficient and practical type-checking based verification approaches which are used in many modern programming languages, and the more heavy-weight automated behavioural verification of software, for example using software model-checking, which has so far only found very limited use in practice, because of the significant verification burden involved.

The idea is, intuitively, to start with the type-based approaches. Essentially, the type-checking verification determines that every function implemented in the program conforms to its specified type, that is, for every input that is legimate

according to the type definition, only output that is legimitate to that definition as well will ever be produced. In traditional type-checking these definitions are defined usualy as sets such that the admissible set of output values does not depend on the particular input values, nor on the execution history.

The idea in rich interface specifications is to drop the latter restriction, i.e., to use generalizations of the "type definitions" (i.e., the interface specifications) which are allowed to make the specified sets of admissible output values dependent not only on the last input value to the function under consideration, but even its complete execution history (for the given execution).

Thus, it is not the aim to consider the internal behaviour of the system component for which the interface specification is written, but only to consider the externally observable behaviour of the component. (In that sense, behavioural interfaces are an instance of the general idea of using abstraction to easen verification.) This core idea has of course been around for quite a long time, some relevant key-words include the concepts of observational equivalence [Abr87], rely-guarantee (or assume-guarantee) specifications (see [BS01] for an overview of some relevant concepts and e.g. [PDH99, CRR02] for the connection to software model-checking) etc.. The underlying formal model for many of these approaches are often state machines where the transitions are labelled with the input and output values (these can take various forms including Mealy and Moore automata [BS01], Interface automata [dAH01], Interface Input/Output Automata [LNW06] etc.). The aim is to verify the software against the state machine in the sense that at each point in the execution history, the admissible output is specified by the output values on the relevant transitions, which depend on the current input value to the function as well as the execution history.

Interface specifications have been used in a number of approaches to software verification, such as the C checker LCLint [EGHT94] (or its successor Splint), the model-checkers SLAM [BR02] and Blast [BCH+04], or the Java interface specification notation JML [BCC+05].

3 Verifying Crypto-protocol Implementations using Interface Specifications

One interesting application domain for the verification using interface specifications is the verification of communication protocol implementations. These protocols are usually specified by reference to the externally observable behavior, where "externally observable" in this case means the messages that are exchanged over the communication network. Such specifications actually do occur "in the wild" (i.e. in industry), for example in the form of message sequence charts or UML sequence diagrams (specifying one possible scenario involving several communication partners) or (UML) Statecharts (specifying each communication partner in isolation but including branching behaviour). This is also a typical situation where, if at all possible, one would not want to be forced to look at the actual implementation details of the protocols, but only consider the externally observable behaviour of the protocol implementation, because the

implementations can be surprisingly complex in many application domains of communication protocols.

Of course, this raises the interesting question which kind of verification is at all possible without having to consider every and all of the messy implementation details. Let us consider some possible alternatives:

Run-time verification (in the sense of "verification" at run-time): The is the most straightforward form of verification using interface specifications. The idea is to derive (possibly automatically) monitors (which may be simple assertion checks) from the interface specification and include them in the implementation code such that at run-time, whenever the behavioural interface specification given by the protocol specfication is violated, the execution will be stopped (for example by throwing an exception). This approach should be relatively easy to implement and be effective, but may induce a performance burden at run-time. Also, the possibility of termination at run-time when the interface spec is violated may be undesirable.

Testing : To remove the two disadvantages that come with run-time verification as defined above, one might try to establish that the assertions that were inserted from the interface specification will in fact never be violated. This would solve not only the second disadvantages of termination at run-time, but then one could again remove the assertions, getting rid of the performance burden. This option is often seen to be the most cost-effective in industry. However, for very high reliability requirements, it will only work when exhaustive testing is possible, which may not be the case in a system which can openly interact with its environment (as has often to be assumed for communication protocols). Also, in the special case of cryptographic protocols, exhaustive testing on the lowest implementation level including the crypto-algorithm details should not be possible (at least not without further tricks based on inspection and abstraction of the source-code), because that might imply the possiblity of a brute-force attack for the adversary.

Static verification : The limitations with testing have of course motivated much work in static software verification, such as software model-checking. However, despite very successful work in this area (including the tools [BR02, BCH$^+$04] mentioned above), the challenge remains that it is difficult to establish a property of an implementation (even a seemingly abstract property such as specificied using interface specifications) without having to consider all the details of the implementations. This includes for example even the implementation of library functions (say in C, the strcmp function). Ideally, of course such functions would be verified against the relevant properties they are supposed to enforce once and for all, and then in the verification of software that uses these functions, one could substitute the implementations of the functions by these abstract properties, which should make verification more efficient. In practice however, we don't yet seem to have such verified properties for library functions readily available for reuse. One possible strategy to deal with this is to abstract away the functions that are being called until finally the code verification tools return a verification result. The

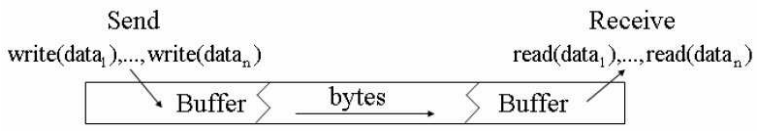


Fig. 1. Communication in crypto protocols such as SSL

other strategy is to start out the with abstract view in the first place, for example by making use of interface specifications (still in the context of static verification).

These considerations give a good motivation to try to apply verification based on interface specification to implementations of protocols (in particular crypto-protocols), although this will still be a challenge in the context of practical verification for the reasons explained above.

In this paper, we report on an approach that aims to do this in the case of Java-based implementations of crypto-protocols. We thus continue with giving some background on how communication in these protocols is often implemented in Java.

To determine the data sent and received, we first deal with the mechanisms which implement a send or a receipt procedure. We assume that each message is represented by a message class. It stores the data to be written in the communication buffer. Conversely, this class can also read messages from the communication buffer (this communication principle is visualized in Figure 1). We found that this mechanism is often (and in particular in the Jessie project discussed in Sect. 5) implemented using methods write() (for sending messages), and read() (for receiving them). Furthermore, the occurrences of the method write() (resp. read()) which are called at the class java.io.OutputStream (resp. java.io.InputStream) is used to identify the individual message parts within the communication procedure in the form of parameters that are delivered or the assignments made. In more detail, communication is implemented as follows: With the method call msg.write(dout, version), the message msg is written into the output buffer dout. Each occurrence of such a method call can be identified and associated with the abstract function send(msg) in the specification model. 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 an example, the code fragment for initializing and sending the ClientHello message is given in Figure 2.

In the remainder of the paper, we will show how to apply the interface specification based verification approach outlined above to the verification of crypto-protocol implementations, following earlier work such as [Jür07]. As interface specifications, we will use UML sequence diagrams such as the example in Fig. 6.

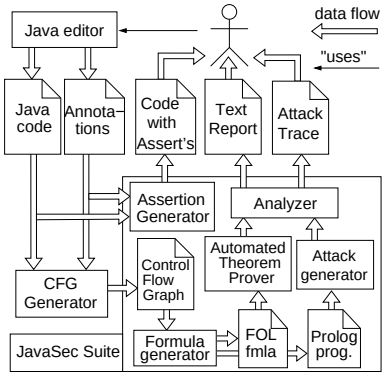
4 Verifying Crypto-protocol Implementations

The analysis approach used here works with the well-known Dolev-Yao adversary model for security analysis. 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 how to link the Java program to the specification model and how to generate the FOL formulas, which are given as input to the ATP. The corresponding tool-flow is shown in Fig. 3a). Because of space limitations, we can not explain all steps required for linking the Java code to the specification model in every technical detail. 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, 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 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. 3b). There, the symbols E, E', and K denote terms inductively constructed in this way. For example, the term ver(E, K, E') denotes the boolean value which signifies whether the verification of the signature E against the plain text E′ using the key K is successful. 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 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. 3b) but are formalized implicitly in the logical formula by introducing new

```
ClientHello clientHello = new ClientHello(session.protocol,clientRandom,sessionId,
                                          session.enabledSuites,comp, extensions);
Handshake msg = new Handshake(Handshake.Type.CLIENT_HELLO, clientHello);
msg.write (dout, version);
```

Fig. 2. Initializing and sending the ClientHello message



- $\mathsf{enc}(E, K)$ (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.)

Fig. 3. a) Tool-flow of the JavaSec suite; b) Abstract crypto operations

```
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)))).
```

Fig. 4. Some general crypto axioms

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 details on this.

The predicates defined to hold for a given system are defined as follows. For each publicly known expression E, the statement knows(E) is given. 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. 4. 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 can be linked to a specification model which gives rise to a logical formula characterizing the interaction between the adversary and the protocol participants (the bottom left part of Fig. 3a). We explain the translation first for a simplified fragment of Java without loops and concurrency. 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 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 get the FOL formula for the program, we first construct the specification model capturing the abstract behaviour of the program. This can for example be done with the help of tools for generating control flows graphs (such as Code Logic), or by constructing it manually from the textual specification of the protocol. The model we construct is a state machine with transitions carrying labels of the form *await message e – check condition g – output message e'*. A state machine 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 state machine 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. 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.

Lastly, we map each 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 the condition g equal to true where they are not needed.

For the mapping from the state machine 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 == by the binary Boolean function $\mathsf{equal}()$ and similarly for the other Boolean connectives.

Suppose now that we are given a transition $l = (\mathsf{source}(l), \mathsf{event}(l), \mathsf{guard}(l), \mathsf{msg}(l), \mathsf{target}(l))$ 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 state machine. For each such transition l, we define a predicate $\mathsf{PRED}(l)$ as in Fig. 5. If a next transition l' does not exist, $\mathsf{PRED}(l)$ is defined by substituting $\mathsf{PRED}(l')$ with true in Fig. 5. The formula formalizes the fact that, if the adversary knows

$$\mathsf{PRED}(l) = \forall exp_1, \ldots, exp_n . \Big(\mathsf{knows}(exp_1) \wedge \ldots \wedge \mathsf{knows}(exp_n) \wedge \ cond(exp_1, \ldots, exp_n) \\ \Rightarrow \mathsf{knows}(exp(exp_1, \ldots, exp_n) \wedge \ \mathsf{PRED}(l') \Big)$$

Fig. 5. Transition predicate

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 state machine is finite and cycle-free. The construction can be refined to allow loops (by using infinite arrays for the variables updated in the loop), recursion, and concurrent threads.

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 state machine 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. 4, 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, $\mathsf{knows}(\mathsf{secret})$ 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 $\mathsf{knows}(\mathsf{secret})$ 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 $\mathsf{knows}(\mathsf{secret})$ from the axioms, this means that the adversary will not get the data represented by secret (relative to our system and adversary model).

5 The Example Application: The SSL project JESSIE

We apply the approach sketched above to the implementation of the Internet security protocol SSL in the project JESSIE, which is an open-source implementation of the Java Secure Sockets Extension (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 [AN96] and is therefore an interesting target for a security analysis. In this paper, we concentrate on the fragment of SSL that uses RSA as the cryptographic algorithm and provides server authentication (cf. Fig. 6).

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.

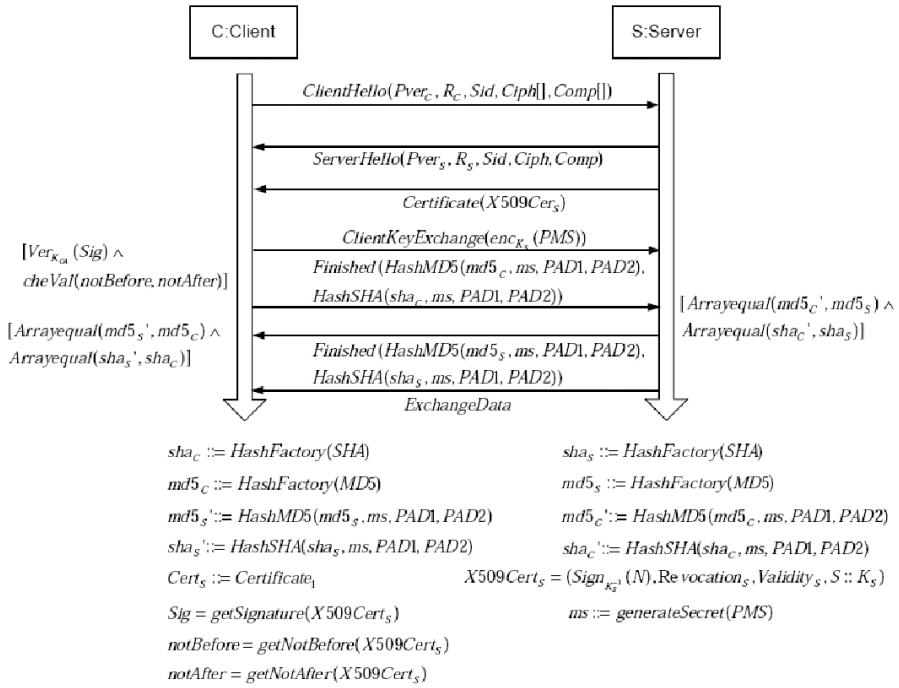


Fig. 6. The cryptographic protocol implemented in SSLSocket.java

To link the code to the specification in Fig. 6, we firstly explain how important elements at the model level are implemented at the implementation level. This is done in the following three steps:

- Step 1: Identification of the data transmitted in the send and receipt procedures at the implementation level.
- Step 2: Interpretation of the transferred data and comparison with the sequence diagram.
- Step 3: Identification and analysis of the cryptographic guards at the implementation level.

In step 1, the communication at the implementation level is examined and determined how the data that is sent and received can be found in the source code. Afterwards, in step 2, a meaning is assigned to the data. The interpreted data elements of the individual messages are then compared with the appropriate elements in the model. In step 3, it is described how one can identify the guards from the model in the source code.

Step 1: In our particular protocol, 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 interaction between client and server that is specified in Figure 6. Each of the messages is implemented by a class, whose main

Message name	Class of Message Type	Message Type
ClientHello	ClientHello	CLIENT_HELLO
ServerHello	ServerHello	SERVER_HELLO
*Certificate**	Certificate	CERTIFICATE
ClientKeyExchange	ClientKeyExchange	CLIENT_KEY_EXCHANGE
Finished	Finished	FINISHED

Fig. 7. Data for the Handshake message

methods are called by the doClientHandshake() rp. doServerHandshake() methods. The associated data is given in Figure 7.

```
Random(int gmtUnixTime, byte[] randomBytes)
  {
    this.gmtUnixTime = gmtUnixTime;
    this.randomBytes = (byte[]) randomBytes.clone();
  }
```

Fig. 8. Constructor for random

Step 2: In order to be able to make a comparison of the implementation with the abstract model, we must first determine for the individual data how it is implemented on the code level, to then be able to verify that this is done correctly. We explain this exemplarily for the variable randomBytes written by the method ClientHello to the message buffer. By inspecting the location at which the variable is written (the method write(randomBytes) in the class Random), we can see that the value of randomBytes is determined by the second parameter of the constructor of this class (see Figure 8).

Therefore the contents of the variable depends on the initialization of the current random object and thus also on the program state. Thus we need to trace back the initialization of the object. In the current program state, the random object was passed on to the ClientHello object by the constructor. This again was delivered at the initialization of the Handshake object in SSLSocket. doClientHandshake() to the constructor of Handshake. Here (within doClientHandshake()), we can find the initialization of the Random object that was passed on. The second parameter is generateSeed() of the class SecureRandom from the package java.security. This call determines the value of randomBytes in the current program state. Thus the value randomBytes is mapped to the model element R_C in the message ClientHello on the model level. For this, the method java.security.SecureRandom.

generateSeed() must be correctly implemented. To increase our confidence in this assumption of an agreement of the implementation with the model (although a full formal verification is not the goal of this paper), all data that is sent and received must be investigated.

As an example, the following assertion for the verification of a hash at the client side in the Jessie doClientHandshake() method is inserted just before the handshake phase is finished successfully:

$$\mathsf{assert}\ \Big(\mathsf{Arrays.equals}(\mathsf{finis.getMD5Hash}(), \mathsf{verify.getMD5Hash}())\ \&\&\ \mathsf{Arrays.equals}(\mathsf{finis.getSHAHash}(), \mathsf{verify.getSHAHash}())\Big)$$

Note that the check that the type of the message received is actually correct is also generated as an assertion as follows, although this is only implicitly contained in our abstract specification.

$$\mathsf{assert}\ \Big(\mathsf{msg.getType}() = \mathsf{Handshake.Type.SERVER\_HELLO}\Big)$$

Step 3: We now explain at the hand of an example how the guards from the SSL specification in Figure 6 can be identified on the code level and how it can be checked that these are in fact correctly implemented. To explain the idea, we concentrate on the Check_Certificate guard from Figure 6. By manually inspecting the source code, one can find the call of a checkServerTrusted method directly after the point where the messages Certificate and Finished are received, which corresponds to the position of the Check_Certificate guard in the model. For the Check_Certificate guard we now explain how one can establish that it corresponds to the guard $\mathsf{ver}(\mathsf{cert}_S)$ on the semantic level and that it is reached within each program run. The investigation shows that first the validity of the individual certificates of the certificate chain peerCerts is queried. Subsequently, each certificate with the key of the predecessor in the certificate chain is verified, until the root is reached. For these it is examined whether it is referred to by one thrust ancor. If not, a CertificateException is thrown, which leads to abort the handshake dialogue. The function doVerify (Signature sig, PublicKey key) is used for verifying a certificate.

We will now explain how the tool automatically verifies, using the annotations defined above, that the code enforces the checks that have to be performed according to the protocol specification. Again, let p be the program point where a protocol message is received and q the point where the next message is sent out. Let g be the condition that according to the specification has to be checked between the program points p and q. Then the tool verifies that the condition g is enforced by the program between the execution points p and q. To verify this, our tool inspects the conditionals and exceptions between p and q to find out whether g is enforced, based on a control flow analysis using control flow graphs, where all paths between the nodes representing the program points p and q have to be examined to see whether they enforce g. As an example, we consider the guard $\mathsf{g} = \mathsf{ver}(\mathsf{cert_S})$ that is performed by the client according to the specification in Fig. 6. We explain in detail how our tool automatically verifies

that the Jessie implementation correctly enforces this guard. The tool makes use of the control flow graph which can be generated from the source code using tools available for this purpose (in our case using the tool CodeLogic).

According to the annotations defined based on the textual specifications, the check g is implemented by the method call session.trustManager.checkServerTrusted(peerCerts, suite.getAuthType()) (which throws an exception if the check fails). By tracking the various write and read calls, the tool also determines where the program points p resp. q are located, where the last message is received resp. the next message is sent out. In this case, the last message before the guard should be checked is received using the command msg = Handshake.read(din, certType) at the program point marked p. This corresponds to the point where the message Certificate($\mathsf{X509Cert_S}$) is received at the Client side in the sequence diagram in Fig. 6. The next message after the guard should be checked is sent out using the command msg.write(dout, version) at the program point q. This corresponds to the point where the message ClientKeyExchange ($\mathsf{enc_{K_S}}$(PMS)) is sent at the Client side in the sequence diagram in Fig. 6. Thus the tool has to check that for each execution of the program, the method call session.trustManager.checkServerTrusted(peerCerts, suite.getAuthType() will be executed between the program points p and q. The tool proves this formally by making use of program reasoning rules. Informally, in this simple example one can see that this is indeed the case by inspecting the control flow graph, where one can see that there is no path from p to q except the one where g is checked, since the jump into the catch-block leads to a termination of the program.

We then 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.

6 Conclusion

We presented the details about an application of automated theorem provers for first order logic in an approach to verify the open-source Java implementation Jessie of the SSL protocol. In particular, we explained how this work can be viewed in the context of verification using behavioural interface specifications.

Future work includes plans to investigate the use of compositional verification techniques, improving on first results such as documented in [Jür06].

Acknowledgements: Help from David Kirscheneder, Haoyang Lin, and Chang Li with the implementation details of Jessie is gratefully acknowledged.

References

[Abr87] S. Abramsky. Observation equivalence as a testing equivalence. *Theor. Comput. Sci.*, 53:225–241, 1987.

[AG99] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, January 1999.

[AN96] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, January 1996.

[BBD+05] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H.R. Nielson. Static validation of security protocols. *Journal of Computer Security*, 13(3):347–390, 2005.

[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. *Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[BCH+04] D. Beyer, A.J. Chlipala, T.A. Henzinger, R. Jhala, and R. Majumdar. The blast query language for software verification. In Roberto Giacobazzi, editor, *SAS*, volume 3148 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2004.

[BFGT06] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. In *CSFW*, pages 139–152. IEEE Computer Society, 2006.

[BR02] T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

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

[CRR02] S. Chaki, S.K. Rajamani, and J. Rehof. Types as models: model checking message-passing programs. In *POPL*, pages 45–57, 2002.

[dAH01] L. de Alfaro and T.A. Henzinger. Interface automata. In *ESEC / SIGSOFT FSE*, pages 109–120, 2001.

[EGHT94] D. Evans, J.V. Guttag, J.J. Horning, and Yang Meng Tan. LCLint: a tool for using specifications to check code. In *SIGSOFT FSE*, pages 87–96, 1994.

[GLP05] Jean Goubault-Larrecq and Fabrice Parrennes. Cryptographic protocol analysis on real c code. In *VMCAI'05*, Lecture Notes in Computer Science. Springer-Verlag, 2005.

[Gor06] Andrew D. Gordon. Provable implementations of security protocols. In *LICS*, pages 345–346. IEEE Computer Society, 2006.

[HALM06] C.L. Heitmeyer, M. Archer, E.I. Leonard, and J.D. McLean. Formal specification and verification of data separation in a separation kernel for an embedded system. In *CCS*, pages 346–355. ACM, 2006.

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

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

[Jür05] J. Jürjens. Sound methods and effective tools for model-based security engineering with UML. In *27th International Conference on Software Engineering (ICSE 2005)*. IEEE Computer Society, 2005.

[Jür06] J. Jürjens. Security analysis of crypto-based Java programs using automated theorem provers. In S. Easterbrook and S. Uchitel, editors, *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*. ACM, 2006.

[Jür07] J. Jürjens. Automated security verification for crypto protocol implementations: Verifying the jessie project. In *Seventh International Workshop on Automated Verification of Critical Systems (AVOCS 2007)*, Oxford, 2007.

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

[LNW06] K.G. Larsen, U. Nyman, and A. Wasowski. Interface input/output automata. In *14th International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 82–97. Springer-Verlag, 2006.

[Low96] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. *Software Concepts and Tools*, 17(3):93–102, 1996.

[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.

[Pau98] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.

[PDH99] C.S. Pasareanu, MB. Dwyer, and M. Huth. Assume-guarantee model checking of software: A comparative case study. In *5th and 6th International SPIN Workshops*, volume 1680 of *Lecture Notes in Computer Science*, pages 168–183. Springer-Verlag, 1999.

[RS98] P. Ryan and S. Schneider. An attack on a recursive authentication protocol. *Information Processing Letters*, 65:7–10, 1998.

[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.

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

[WBH+02] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topić. Spass version 2.0. In *CADE-18*, volume 2392 of *Lecture Notes in Computer Science*, pages 275–279. Springer-Verlag, 2002.