Verifying an implementation of SSH

Erik Poll[1]* and Aleksy Schubert[1,2]**

[1] Radboud University Nijmegen, the Netherlands
[2] Warsaw University, Poland

Abstract. We present a case study in the formal verification of an open source Java implementation of SSH. We discuss the security flaws we found and fixed by means of formal specification and verification – using the specification language JML and the program verification tool ESC/Java2 – and by more basic manual code inspection. Of more general interest is the methodology we propose to formalise security protocols such as SSH using finite state machines. This provides a precise but accessible formal specification, that is not only useful for formal verification, but also for development, testing, and for clarification of official specification in natural language.

1 Introduction

The past decade has seen great progress in the field of formal analysis of security protocols. However, there has been little work or progress on verifying actual *implementations* of security protocols. Still, this is an important issue, because bugs can make an implementation of a secure protocol completely insecure. A fundamental challenge here is posed by the big gaps between (i) the official specification of a security protocol, typically in natural language; (ii) any models of (parts of) the protocol developed for formal verification of security properties, e.g. using model checking; and (iii) actual implementations of the protocol. In an effort to bridge these gaps, we performed a case study in the formal specification and verification of a Java implementation of SSH. We considered an existing implementation, MIDP-SSH[3], which is an actively maintained open source implementation for use on Java-enabled mobile phones. MIDP-SSH is a typical implementation in the sense that it is not written from scratch but based on an earlier one, re-using code from a variety of sources.

In order to express the properties to be verified for the source code, we used the Java Modeling Language (JML) [9]. JML is a specification language designed to describe properties of Java programs. It supports all the important features of the Java language e.g. inheritance, subtyping, exceptions etc. JML is supported by a range of tools for dynamic or static checking; for an overview see [2]. We used the extended static checker ESC/Java2 [3], the successor of ESC/Java [4]. This

* Supported by the Sixth Framework Programme of the EU under the MOBIUS project FP6-015905.
** Supported by the Sixth Framework Programme of the EU under the SOJOURN project MEIF-CT-2005-024306.
[3] Available from `http://www.xk72.com/midpssh/`.

tool tries to verify automatically JML-annotated source code, using a weakest precondition calculus and an automated theorem prover.

The structure of the paper Section 2 describes the informal code inspection carried out as first stage of our analysis, and Section 3 describes the more formal approach taken after that. Section 1.1 below presents an overview of the approach taken in these two stages. We draw our conclusions and discuss possible future work in Section 4.

1.1 Methodology

After considering the security requirements of the application, our analysis of the implementation proceeded in several steps.

The first stage, described in Section 2, was an ad-hoc manual inspection of the source code. We familiarised ourselves with the design of the application, considered which parts of the code are security-sensitive, and looked for possible weaknesses. This led to discovery of some common mistakes – or at least bad practices which should be avoided in security-sensitive applications.

The next stage, described in Section 3, involved the use of the formal specification language JML and the program verification tool ESC/Java2. Here we can distinguish two steps:

- The first step, discussed in Section 3.1, was the standard one when using ESC/Java2: we used the tool to verify that the implementation does not throw any runtime exceptions. For instance, the implementation might throw an ArrayIndexOutOfBoundsException due to incorrect handling of some malformed data packet it receives. This step revealed some bugs in the implementation, where sanity checks on well-formedness of the data packets received were not properly carried out. This would only allow a DoS attack, by making the SSH client crash on such a malformed packet. Of course, for an implementation in a type-unsafe language such as C, as opposed to Java, these bugs would be much more serious, as potential sources of buffer overflow attacks.
  The process of using ESC/Java2 to verify that no runtime exceptions can occur, incl. the process of adding the JML annotations this requires, forces one to thoroughly inspect and understand the code. As a side effect of this we already spotted one serious security flaw in the implementation.
- The next step, discussed in Section 3.2, was to verify that the Java code correctly implements the SSH protocol as officially specified in RFCs 4250-4254 [16, 14, 17, 15]. This required some formal specification of SSH. For this we developed our own formal specification of SSH, in the form of a finite state machine (FSM) which describes how the state of the protocol changes in response to the different messages it can receive. This is of course only a *partial* specification, as it specifies the *order* of messages but not their precise format. Still, it turned out to be interesting enough, as we hope to demonstrate in this paper.

This last step of the verification is probably the most interesting. Firstly, we found that obtaining the finite state machine from the natural language description in the RFCs was far from trivial, and it revealed some ambiguities and unclarities. It is not always clear what the response to an unexpected, unsupported or simply malformed message should be: some of these *may* or *should* be ignored but others *must* lead to disconnection. Secondly, verifying that the implementation meets this partial specification as given by the FSM revealed some serious security flaws in the implementation. In particular, the implementation is vulnerable to a man-in-the-middle attack, where an attacker can request the username and password of the user *before* any authentication has taken place and before a session key has been established. A secure implementation should of course never handle such a request.

2 Stage 1: Informal, ad-hoc analysis

Prior to any systematic analysis of the application as discussed in the next section, we read the security analysis of the SSH protocol provided in the RFCs [16]. Then we extended the analysis to cover the issues closely related to the Java programming language and to the Java MIDP platform. We located the part of the source code which directly implements the protocol and tried to relate the results of the security analysis to the source code, but without trying to understand the logic of the implementation. In the course of these steps, we already spotted some (potential) security problems. Here is a description of the most important ones:

Weak/no authentication The SSH client does not store public key information for subsequent sessions: it will connect to any site and simply ask that site for its public keys, without checking this against earlier runs and asking the user to accept a new or changed public key. In other words, there is no real authentication before starting an SSH session. This is especially strange as the application stores certain session related information (i.e. host name, user name, and even password) in the MIDP permanent storage – record stores.

There is a countermeasure that allows the user to authenticate the server she or he is connecting to: the SSH client displays an MD5 hash of the server's public key as 'fingerprint' of the server it connects to. The user can check to see if this MD5 hash has the right value. Of course, the typical user will not check this.

Note that unauthenticated key exchange is a well-known and common security mistake; it is for instance listed in [6]. This highlights the importance that programmers are aware of such common mistakes!

Poor use of Java access restrictions The implementation does not make optimal use of the possibilities that Java offers to restrict access to data, with the visibility modifiers, such as `public` and `private`, and the modifier `final` to make fields immutable.

For instance, the implementation creates an instance of the standard library class `java.lang.Random` for random number generation. The reference to this object is stored in a *public* static field `rnd`. Untrusted code could simply modify this field, so that it for instance points to an instance of `java.lang.Random` with a known seed, or to an instance of some completely bogus subclass of `java.lang.Random` which does not produce random numbers at all. The field `rnd` should be `private` or `final` – or, better still, both – to rule out such tampering.

In all fairness, we should point out that for the current version of the MIDP platform the threat of some hostile application attacking the SSH client by changing its public fields is not realistic. A restriction of the MIDP platform is that at most one application – or *midlet*, as applications for the MIDP platform are called – is running at the same time, so a hostile application cannot be executing concurrently with the SSH midlet. Moreover, each time the SSH client is started it will initialise its fields from scratch. Still, such restrictions are likely to be loosened in the future, and the code of MIDP-SSH might be re-used in applications for other Java platforms where these restrictions do not apply.

A similar problem occurs with the storage of the contents of P- and S-boxes in the implementation. The class `Blowfish` in the implementation uses an array

```
final static int[] blowfish_sbox = { 0xd1310ba6, ... };
```

This integer array is final, so cannot be modified. However, the *content* of the array is still modifiable. The field has default package visibility, which gives rather weak restrictions about who can modify is, as explained in [10], so hostile code could modify the S-boxes used by the SSH client, and at least create a DoS attack. The field should really be private and there is no reason why it cannot be. Again, for the MIDP platform this is not really a threat, due to its restrictions discussed above.

Checking if access modifiers can be tightened need not be done manually, but can be automated, for instance using JAMIT[4]. The problems in the application suggest that systematic use of such a tool would be worthwhile.

Control characters One of the security threats mentioned in the security analysis is the scenario when a malicious party sends a stream of control characters which erases certain messages to lure the user into performing an insecure action. Although the SSH client does interpret some control characters, there is no operation to ensure that only safe control sequences appear on the user's terminal.

Downloading of the session information The application implements functionality to download a description of an SSH session to execute. Such a description can contain the information about the user and a host name. The transfer of such information over the network in cleartext is an obvious compromise of the security as third parties can associate the login with the machine. Moreover,

[4] See `http://grothoff.org/christian/xtc/jamit/`

data downloaded in this way is not displayed to the user who demanded it. In this way it is easy to realize a spoofing attack which forwards the user to a fake SSH server which steals the password.

3 Formal, systematic analysis using JML and ESC/Java2

The analysis using more formal methods consisted of two stages. The first stage was to verify that the implementation does not throw any runtime exceptions, e.g. due to null pointers, bad type casts, or accesses outside array bounds. The second one was to (partially) specify SSH, by means of a finite state machine, and verify that the implementation correctly implements this behaviour.

3.1 Stage 2: Exception Freeness

The standard first step in using ESC/Java2 is to check that the program does not produce any runtime exceptions. Indeed, often this is the only property one checks for the code. Although it is a relatively weak property, verifying it can reveal quite a number of bugs and can expose many implicit assumptions in the code. Just establishing exception freeness requires the formalisation of many properties about the code, as JML preconditions, invariants, and sometimes postconditions. For instance, invariants that certain reference fields cannot be null are needed to rule out NullPointerExceptions, and invariants that certain integer fields are not negative or have some maximum value are needed to rule out ArrayIndexOutOfBoundsExceptions.

Like any verification with ESC/Java2, checking the absence of exceptions relies on the axiomatisation of Java semantics built into the tool and on specifications of any APIs used, e.g. for library calls such as `System.arraycopy`, which are given in a standard set of files with JML specifications for core API classes. Correctness of the results of ESC/Java2 relies on the correctness of this axiomatisation and these API specifications.

Trying to check that no runtime exceptions occur with ESC/Java2 revealed some bugs in the implementation, namely missing sanity checks on the well-formedness of the data packets before these packets are processed. This means that the SSH client could crash with an ArrayIndexOutOfBoundsException when receiving certain malformed packets. Such Denial-of-Service attacks are discussed in the RCFs.

The process of using ESC/Java2 to check that no runtime exceptions can occur – incl. the adding of all the JML annotations this requires – forces one to thoroughly inspect and understand the code. As a side effect of this we spotted a serious security weakness in the implementation, namely that it does not check the MAC of the incoming messages, so it is vulnerable to certain replay attacks.

The whole process of proving exception freeness, including fixing the code where required, took about two weeks.

3.2 Stage 3: Protocol specification and verification

In addition to just proving that the implementation does not throw runtime exceptions, we also wanted to verify that it is a correct implementation of the client side of the SSH protocol, as specified in the RFCs. This requires some formal specification of SSH, of course.

Formal specification of SSH as FSM Unfortunately we could not find any formal description of SSH in the literature; the only formal description we could find [12] only deals with a part of the whole SSH protocol. Therefore we developed our own formal specification of SSH, in the form of a finite state machine (FSM) which describes how the state of the protocol changes in response to the different messages it can receive. This is of course only a *partial* specification, as it only specifies the *order* of messages but not their precise format. Still, this partial spec was interesting enough, as we hope to demonstrate in this paper.

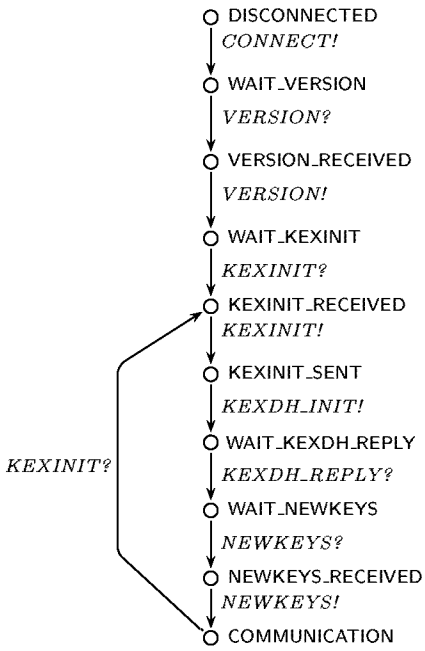

Fig. 1. A simplified view of the FSM specifying the behaviour of the SSH client, without optional features described in the RFCs that are not supported, and ignoring the aspects described in Fig. 3. The names of the transitions are the same names used in the RFCs. Labels ending with ! are outputs of the client to the server, labels ending with ? denote inputs to the client.

It turns out that the SSH protocol involves about 15 kinds of messages and its session key negotiation phase has about 20 different states. One complication in defining an FSM describing the client side behaviour of the protocol is that the SSH specifications present the protocol as a set of features which are partly obligatory and partly optional. A FSM that includes all these optional parts is given in Fig. 2. For simplicity, we focused our attention on those parts of the protocol that this particular implementation actually supports. This simplifies the overall behaviour, namely to that shown in Fig. 1. This behaviour corresponds to the left-most branch in the full specification of SSH given in Fig. 2.

Fig. 1 not only ignores options not implemented, but also includes an apparently common choice made in the implementation that is left open by the official specification. Section 4.2 of [17] states: "*When the connection has been established, both sides MUST send an identification string*". This specifies that both client and server must send an identification string, but does not specify the order in which they do this. In principle, it is possible for both sides to wait for the other to send the identification string first, leading to deadlock. The MIDP-SSH implementation chooses to let the client wait for an identification string from the server (the transition *VERSION?* in Fig. 1) before replying with an identification string (the subsequent transition *VERSION!*). This appears to be the standard way of implementing this: OpenSSH makes the same choice. In fact, an earlier specification of SSH 1.5 [13, Overview of the Protocol] does prescribe this order; it is not clear to us why the newer specification [17] does not. Moreover, it is not clear if this is a deliberate underspecification or a mistake. Of course, one of the benefits of formalising specifications is that such issues come to light.

Fig. 1 does not tell the whole story, though. It only specifies the standard, correct sequence of messages, but does not specify how the client should react to unexpected, unsupported, or simply malformed messages. This is where much of the complication lies: some of these messages *may* or *should* be ignored, but others *must* lead to disconnection. Adding all the transitions for this to Fig. 1, (or, worse still to Fig. 2) would lead to a very complicated FSM that is hard to draw or understand, and very easy to get wrong. We therefore chose to specify these aspects in a separate FSM, given in Fig. 3.

The SSH specification states that after the protocol version is negotiated, i.e. from the state WAIT_KEXINIT onwards, the client should always be able to handle a few messages in a generic way. Some of these messages should be completely ignored; some should lead to an *UNIMPLEMENTED!* reply, meaning the client does not support this message; some should lead to disconnection. This aspect is specified in a separate FSM: in the state WAIT_KEXINIT and any later state, the client should implement the additional transitions given in Fig. 3.

In Fig. 3 we use a few additional ad-hoc conventions to keep the diagram readable. ***FOREIGN_MSGS?*** stands for any message that is not explicitly known by the application. As noted above, all such messages should trigger the sending of the *UNIMPLEMENTED* message. Similarly, ***OTHER_KNOWN_MSGS?*** stands for any message that is known, but arrived in a wrong state – these

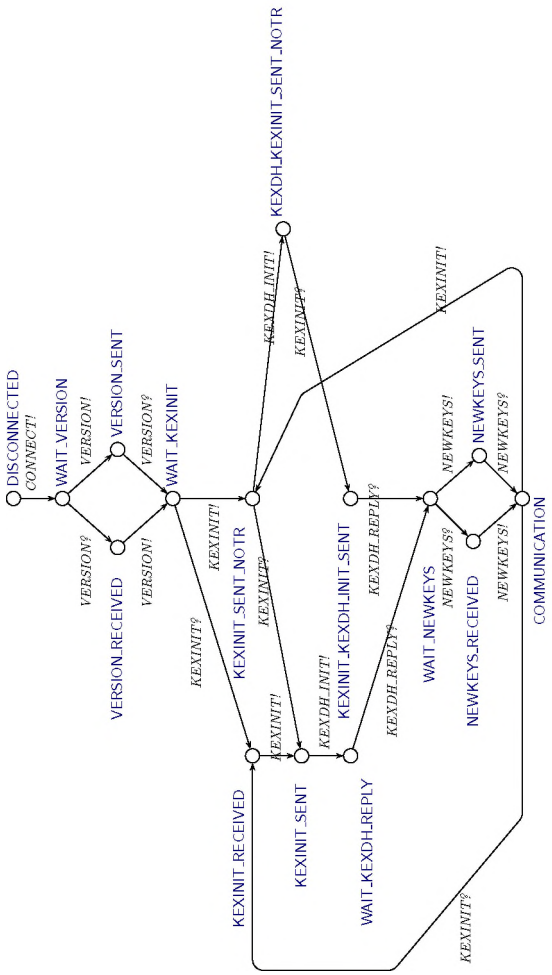

Fig. 2. Specification of the full client side behaviour of SSH, including all optional features. This diagram must still be extended with the additional aspects as given by the diagrams in Fig. 3. *[NB the diagram above have been corrected and simplified from the original WITS'07 publication.]*

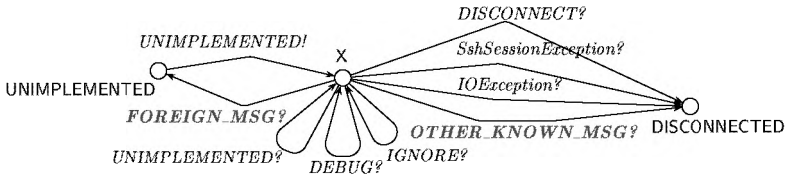

Fig. 3. Additional possible transitions from WAIT_KEXINIT onwards. X stands for any state from WAIT_KEXINIT onwards.

messages lead to disconnection. This diagram is still a simplification because in some states certain known messages should be ignored rather than lead to disconnection, but we do not have space to discuss these details here.

Another ad-hoc convention are the labels *SshException?* and *IOException?*. These transitions represent two exceptional situations that can occur. Firstly, there is the possibility of an IO error (e.g. because the network or the server goes down), which is modelled by the *IOException?* transition. Secondly, there is the possibility that the incoming packet is of a known type but fails to meet the format specified in the RFCs (e.g. the value of the length field exceeds the size of the packet, or the MAC is incorrect), which is modelled by the *SshException?* transition. As you may have guessed, the names of these transitions are inspired the Java exceptions used in the implementation.

Discussion The finite state machines specifying SSH are implicit in the natural language specifications given in the RFCs, but were not so easy to extract, and highlighted some unclarities. We already mentioned the issue that description of the order of certain messages from client to server and back can be interpreted in several ways.

Whereas the names of various types of messages are well-standardised, and we use these in our diagrams, there is *no* explicit notion of state in the SSH specifications. So the names of the states in the diagrams are our invention. This lack of an explicit notion of state is a source of unclarity in the specification. In particular, [16, Sect. 9.3.5] asserts:

> If transmission errors or message manipulation occur, the connection is closed. The connection SHOULD be re-established if this occurs.

but it is hard to figure out which messages should be regarded as message manipulation at a given stage. The RFCs specify forbidden messages in several places, e.g. in [17, Sect. 7.1], e.g.

> Once a party has sent a SSH_MSG_KEXINIT message [...], until it has sent a SSH_MSG_NEWKEYS message (Section 7.3), it MUST NOT send any messages other than: [...]

but it is not obvious that messages other than those listed should be considered as 'manipulations' at this stage.

It would be better if the information about which messages are allowed, can be ignored, or must lead to disconnection in a given state is available in a more structured way. Now this information is spread out over several places in the RFCs. An alternative to using FSMs might simply be a table of states and messages.

Another source of unclarity is the way the standard keywords are used in the specifications. There is an IETF standard which precisely defines the precise meaning of terms such as 'MUST', 'MAY', 'RECOMMENDED', and 'OPTIONAL' [1], but the SSH specification is not consistent in using these keywords. For example, [17, Section 4] says

> Key exchange *will* begin immediately after sending this identifier.

which presumably means that it 'MUST' (and that any other behaviour 'MUST' be considered as manipulation and lead to disconnection?).

Finally, in [17, Section 6] we noted that it is not clear if a well-formed packet may have a zero-length payload section or if such a packet should always be treated as malformed, because it is impossible to determine its type, which is crucial for any handling of the packet. (The specification does not forbid such packets, but for instance OpenSSH treats them as an error and quits the client).

3.3 Verification of MIDP-SSH

Before we even attempted a formal verification that the MIDP-SSH correctly implements the specification as given by the FSMs, it was easy to see that the implementation was not correct: it did not correctly record the protocol state, and it accepted and processed many messages which following the FSMs should lead to disconnection. The prime example of this was that a request for username and password would be processed by the SSH client in any state.

Therefore we improved the implementation before attempting formal verification: we re-factored the code so the handling of each message was done by a separate method, we improved the recording of the protocol state, and we added case distinctions based on the protocol state to obtain the right behaviour in each state.

To verify that the software correctly implemented the finite state machine, we then used AutoJML[5] [7], a tool that generates JML specifications (or Java code) from finite state machines. This tool had to be adapted to cope with our use of several state diagrams to express various aspects of the behaviour, i.e. with Fig 3 expressing aspects of the behaviour that should be added to the overall behaviour in Fig. 1. (The alternative would have been to draw the very large finite state machine that would result from adding these aspects to the overall behaviour in Fig. 1.)

[5] Available from `http://autojml.sourceforge.net`

We added the specifications generated by AutoJML to the source code and verified them using ESC/Java2. This revealed there were still errors in the (already improved) implementation, where certain methods handled incoming messages in a different way than prescribed by the FSM. Even though we were aware that the handling of exceptions is a delicate matter and paid particular attention to this, we still missed updates to the internal state variable in certain cases when the exceptions were thrown.

4 Conclusions

Now that there are various mature tools available to verify security properties of abstract security protocols, we believe it is time to tackle the next challenge, namely trying to verify the security of real implementations of such protocols.

This paper reports on an experiment to see if and how formal methods – in particular formal specification using finite state machines, the specification language JML, and the program checker ESC/Java2 – can be used for to verify an existing Java implementation of SSH. In the end, we managed to verify the implementation in the sense that it never throws an exception (which is maybe more a safety property than a security property) and that it correctly implements the SSH protocol as specified by finite state machines that we developed as formalisation of the official SSH specifications. Along the way we found and fixed several security flaws in the code. Some of these were found as a direct consequence of the verification, some were found more as a side-effect of having to thoroughly inspect and annotate the code to get it to verify. Using of an extended static checker such as ESC/Java2 is a way to force a very thorough code inspection.

A general conclusion about our case study is that a formal specification of a security protocol that captures all or at least most of the complexities in some format that is readable to implementors is very useful. Given the complexity of real-life protocols, it is easy to get something wrong, as witnessed by the implementation we looked at. The specification of SSH as a finite state machine is formal, but still easy to understand by non-experts. (We are investigating other notations to use instead of finite state machines – more on that below.) We believe that providing such a description as part of official specification would be valuable, as it clarifies the specification and is also useful for development. Indeed, note that anyone who implements SSH will, as part of the work, have to implement a finite state machine that is described in the prose of the SSH RFCs and hence will have to re-do much of the work that we have done in coming up with the description of SSH as finite state machine.

The size of the SSH code we verified (just the code for the protocol, excluding the code for the GUI etc.) is around 4.5 kloc. The whole verification effort took about 6 weeks, including the time it took to understand and formalise the SSH specs, which was about 2 weeks. For widely used implementations of security protocols, say the implementation of SSL in the Java API, such an effort might be considered acceptable.

The second stage in our approach, ensuring the absence of runtime exceptions, can catch programming errors in the handling of individual messages, especially malformed ones. The third stage, verification of conformance to the FSM, can catch programming errors in the handling of sequences of messages, especially incorrect ones. Note that this complements conventional testing: testing – or, indeed, normal use of the application – is likely to reveal bugs in the handling of correctly formatted messages and correct sequences of such messages, but is less likely to reveal bugs in the handling of incorrectly formatted messages or incorrect sequences of messages, simply due to the limitless number of possibilities for this. So our approach may detect errors that are hard to find using testing.

A more practical issue is what the most convenient formalism or format for such finite state machines is, and which tools can be used to develop them. We developed our diagrams on paper and whiteboards, but with large number of arrows this becomes very cumbersome without some ad-hoc conventions and abbreviations. Maybe a purely graphical language is not the most convenient in the long run. Given the complexity of a real-life protocols, some way of separating different aspects in different finite state machines (as we have done with Fig. 1 describing the 'normal' scenario and Fig. 3 describing 'other' scenarios) seems important.

Related work An earlier paper [7] already investigated how a provably correct implementation could be obtained from an abstract security protocol for a very simple protocol. The AutoJML tool we used to produce JML specifications from the finite state machines can also produce a skeleton Java implementation. When developing an implementation for SSH from scratch, rather than examining an existing one as we did, this approach might be preferable. There are already efforts to generate code from abstract protocol descriptions, e.g. to generate Java code from security protocols described in the Spi calculus [11], or to refine abstract state machine (ASM) specifications to Java code [5].

Jürjens in [8] showed how to verify the security of UML models of security related protocols. These UML models are on the level of abstraction similar to the one employed by us in FSMs.

Future work It would be interesting to repeat the experiment we have done for other implementations and for other protocols, i.e. trying to formalise other protocols using FSMs or other formalisms, and using these to check implementations. Of course, for an implementation that is not in Java, but say in C or C++, we might not have program checkers like ESC/Java2. Still, a formalisation of a security protocol is not only useful for program verification, but also as aid to the implementor, as aid for a human code inspection and for testing. Indeed, model-based testing could be used to test if an implementation of SSH conforms to our formal specification of the protocol.

In the end we only verified that the code correctly implements the protocol as described by the finite state machine, not that this protocol is secure, i.e. that it ensures authentication, integrity and confidentiality. Verifying that the

full SSH protocol as described in Fig. 2 meets its security goals still seems an interesting challenge to the security protocol verification community.

As an alternative to using finite state machines, we are currently experimenting with a notation that is similar to the standard format used to describe security protocols, but extended with branching and jump statements. This then allows us not only to specify the normal protocol run, but also how any deviations from the normal protocol run have to be handled. Such a formalism may be more practical notation than a graphical one such as finite state machines, which can become unwieldy, and has the advantage of being closer to conventional formal notation for security protocols.

References

1. S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119, The Internet Engineering Task Force, Network Working Group, March 1997.
2. 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. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, 2005.
3. David R. Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In G. Barthe et.al., editor, *CASSIS 2004*, number 3362 in LNCS. Springer, 2004.
4. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI'02*, pages 234–245, New York, NY, USA, 2002. ACM Press.
5. Holger Grandy, Dominik Haneberg, Kurt Stenzel, and Wolfgang Reif. Developing provable secure m-commerce applications. In *Emerging Trends in Information and Communication Security*, volume 2995 of *LNCS*, pages 115–129, 2006.
6. Michael Howard, David leBlanc, and John Viega. *19 Deadly Sins of Software Security*. McGraw-Hill, 2005.
7. E.-M.G.M. Hubbers, M.D. Oostdijk, and E. Poll. Implementing a formally verifiable security protocol in Java Card. In *Security in Pervasive Computing, SPC'03*, volume 2802 of *LNCS*, pages 213–226. Springer-Verlag, 2004.
8. Jan Jürjens. Sound methods and effective tools for model-based security engineering with uml. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 322–331, 2005.
9. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *Behavioral Specifications of Businesses and Systems*, chapter JML: A Notation for Detailed Design, pages 175–188. Kluwer, 1999.
10. Gary McGraw and Ed Felten. *Securing Java*. Wiley, 1999. Available online at `www.securingjava.org`.
11. Benjamin Tobler and Andrew Hutchison. Generating Network Security Protocol Implementations from Formal Specifications. In E. Nardelli et.al., editor, *IFIP World Computer Congress - Certification and Security in Inter-Organizational E-Services (CSES)*, 2004.
12. David von Oheimb. Formal specification of the SSH transport layer protocol in HLPSL, 2004. Available online at `http://www.avispa-project.org/library/ssh-transport.html`.
13. T. Ylönen. The SSH (Secure Shell) Remote Login Protocol. Internet draft, The Internet Engineering Task Force, Network Working Group, NOV 1995. Available at `http://www.snailbook.com/docs/protocol-1.5.txt`.

14. T. Ylönen. The Secure Shell (SSH) Authentication Protocol. RFC 4252, The Internet Engineering Task Force, Network Working Group, January 2006.
15. T. Ylönen. The Secure Shell (SSH) Connection Protocol. RFC 4254, The Internet Engineering Task Force, Network Working Group, January 2006.
16. T. Ylönen. The Secure Shell (SSH) Protocol Architecture. RFC 4251, The Internet Engineering Task Force, Network Working Group, January 2006.
17. T. Ylönen. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253, The Internet Engineering Task Force, Network Working Group, January 2006.