Protocol state machines and session languages: specification, implementation, and security flaws

Erik Poll and Joeri de Ruiter
Digital Security group, Radboud University Nijmegen
Nijmegen, the Netherlands
Email: {erikpoll,joeri}@cs.ru.nl
Aleksy Schubert
Institute of Informatics, University of Warsaw
Warsaw, Poland
Email: alx@mimuw.edu.pl

***Abstract*—Input languages, which describe the set of valid inputs an application has to handle, play a central role in language-theoretic security, in recognition of the fact that overly complex, sloppily specified, or incorrectly implemented input languages are the root cause of many security vulnerabilities.**

Often an input language not only involves a language of individual messages, but also some *protocol* with a notion of a *session*, i.e. a *sequence* of messages that makes up a dialogue between two parties. This paper takes a closer look at languages for such sessions, when it comes to specification, implementation, and testing – and as a source of insecurity.

We show that these 'session' languages are often poorly specified and that errors in implementing them can cause security problems. As a way to improve this situation, we discuss the possibility to automatically infer formal specifications of such languages, in the form of protocol state machines, from implementations by black box testing.

***Keywords*-protocol state machine; language-theoretic security; formal specification; fuzzing; reverse engineering**

I. Introduction

Protocol state machines, by which we mean finite state machines or automata that describe the message sequences that can occur as sessions for some protocol, have played an important role in much of the security research we have done over the years. This research investigated the security of software for smartcards, incl. bank cards and e-passports [1], [2], [3], [4], feature phone midlets [5], a hardware security token for internet banking [6], [7], and networking protocols such as SSH [8] and SSL/TLS [9]. This paper gives an overview of the research direction that evolved here, where state machines are used to analyse the security of software.

Motivation for writing this paper was the realisation that the use of protocol state machines to specify session languages fits nicely with the ideas behind language-theoretic security. After all, as soon as the interaction between two systems not just involves some data format but also some protocol, then the input language effectively consists of two levels: a language of *messages*, which describes the data format of individual messages sent from one party to the other, and a language of *sessions*, or message sequences, which describes valid dialogues between two parties.[1]

Where much of the work on language-theoretic security focuses on the former level (and rightly so, as this is where most of the security problems arise), the focus of this paper is on the latter, i.e. on session languages. We explore the differences between languages of messages and languages of sessions, argue that the languages of protocol sessions can and should be more explicitly and formally specified, and show examples of security vulnerabilities that are caused by incorrectly implementing them. We also discuss the possibilities of automatically inferring protocol state machines from implementations. Algorithms to do this are known from automata theory, notably Angluin's L* algorithm [10], and available in libraries such as LearnLib [11]. Especially in the absence of clear specifications, this can be a useful first step in taking a more rigorous and structured approach to session languages.

Of course, recognising the importance of rigorously specifying, implementing, verifying, and testing protocols is nothing new. For an interesting historical overview of work on protocol engineering in the 1970s and 1980s we refer to [12].

II. Session languages

Many protocols involve the notion of a session, a sequence of messages exchanged between two or more parties following some standard pattern. For example, security protocols such as SSH or SSL/TLS rely on very specific sequences of messages to establish shared keys and then use these. Implementing such a protocol then not only involves parsing and interpreting individual messages, but also keeping track of the order of these messages and checking some interdependencies. Specifying a protocol therefore not only

[1]Note that it is somewhat confusing to call the session language an input language, as it involves both inputs and outputs.

involves specification of the language of messages, but also specification of the session language.

Most protocols have a so-called *happy flow*, i.e. a normal sequence of messages that happen in most or all 'correct' sessions. But even if the session language is effectively just a single sequence of messages, any implementation will have to cope with errors that deviate from this happy flow, and do so in a right – and secure – way. Here errors can occur in an individual message (e.g., in a cryptographic protocol, a message with an incorrect MAC) or in the order in which messages are received.

To handle errors a common pattern is that some errors will cause the session to be aborted, while others are simply ignored. Ignoring a message can happen silently, or result in an error message to warn the other party that a message was ignored. Which approach is taken depends on the type of protocol, and the potential harmfulness of the message. For example, any error in the critical phase of a security protocol (say, during key negotiation) should typically lead to this phase being aborted and restarted, as these protocols are notoriously fragile: a small deviation from the correct protocol run may completely destroy all the security guarantees that the protocol is meant to provide.

Apart from handling errors, protocols often include correct messages that may be inserted at any stage. One example is the by now infamous `HEARTBEAT` message in TLS, which can be sent at any time during a connection when no handshake is being performed [13]. Other examples, for SSH, are the `SSH_MSG_IGNORE` message, which can be sent at any moment but should always be ignored (this message can be used for traffic padding, to defeat traffic analysis) and the `SSH_MSG_DEBUG` message which is included for debugging purposes.

Restrictions on the order of messages also arise in many situations where there is some form of access control. For example, often a smartcard will only perform a security-critical operation *after* it has received the correct PIN code, and then only once. Many security requirements can in fact be expressed by constraints on the order of messages using, for example, temporal logic. Indeed, state machines (or automata) have been proposed as a general framework to define categories of security policies [14], namely those that can be enforced by runtime monitoring.

III. Implementing session languages

When it comes to implementing a protocol, there are fundamental differences between message languages and session languages.

Handling an individual incoming message can be cleanly done in two separate stages: first parsing the message and then processing the resulting parse tree. For the first stage one would ideally use a parser generated from a formal grammar. Of course, in practice these stages are often not so cleanly separated, but mixed together in a 'shotgun parser' [15].

Handling the session language is messier, as it has to be done incrementally, one message at a time. We cannot wait for the entire session to be completed, and then feed that whole string to a parser for the session language. If we have a formal specification of the session language, we might be able to generate some code from that, but this then typically results in a skeleton of code that still has to be manually refined to include the required functionality.

There are different ways in which an implementation can keep track of the protocol session. A program can include one or more variables to explicitly keep track of the protocol state. Alternatively, a program can simply use the program point to provide information of the protocol state; this is what happens if we use sequential composition to compose actions.

To illustrate these two approaches, suppose the session of some protocol is always `A;B;C;D`, where `A` and `C` are inputs and `B` and `D` the resulting outputs. This might then be implemented as

```
receiveA();sendB();receiveC();sendD();
```

where the program point then keeps track of the session state. Of course, the implementation should not forget how to cope with incorrect sequences of inputs, for example by throwing an exception when receiving an unexpected input. Alternatively, we could use some state variable `state`, initialised to say 0, and implement the protocol as a repetition of the procedure below

```
step() {
   receiveMsg(in);
   if (in==A && state==0)
     { sendB(); state=1; }
   else if (in==C && state==1)
     { sendD(); state=2; }
   else
     { ...// raise error }
}
```

Here there is a more explicit use of a state machine in the implementation.

Of course, implementations can combine the approaches above, or use more advanced ways to implement the desired control flow. For example, OpenSSH, which is written in C, uses a global array with 256 function pointers to track the protocol state. To process a new incoming message, this array is used to jump to the right procedure to handle that message, based on one byte in the SSH packet that indicates the type of the packet. The contents of the global array are updated at various stages, to change the protocol state. This is an efficient way to implement a state machine, and arguably a clever use of function pointers, but trying to understand the behaviour from the code is far from trivial.

Doing a security code review [16] of OpenSSH, we did confirm that OpenSSH followed the RFCs. To do this, we first drew protocol state diagrams, as discussed in the next section, based on the RFCs. Then, when reading the source code, we used these as a reference to understand and check what the code did. We could not contemplate checking the correctness of an implementation like OpenSSH without drawing the protocol state machines.

IV. Specification of session languages

The session language of a protocol is commonly specified in prose. Sometimes message flow diagrams or message sequence charts are added, but usually only as examples. Such diagrams typically abstract over the actual payloads of messages and only look at the types of these messages. More importantly, they usually only consider the 'happy flow'.

When protocols are more complicated, and they for instance include several happy flows, session languages can be more conveniently – and precisely – described by finite state machines. (Instead of a finite state machine one could of course use a grammar or, in simple cases, a regular expression. However, a description with a state machine is usually more natural, because the notion of 'state' often captures more information than just which subsequent sequences of messages are accepted. So it may make sense to distinguish two states even though they accept the same language.)

Apart from the improved precision and clarity, another advantage of a protocol state machine is that it provides a first step towards an implementation, namely one which uses a program variable to track the protocol state, as discussed above. For this is it is useful if protocol states are named or numbered. Protocol specifications are often meticulous in defining a naming or numbering scheme for the different message types, but often do not introduce any names for protocol states.

We have noticed that in typical protocol specifications the session language is less likely to be rigorously defined than the message language. Most specifications will have an appendix with a BNF grammar to define the format of messages[2], but many only use prose to describe the session language. One factor explaining this, at least for RFCs, may be that RFCs are in ASCII, so including a state machine is tricky. Still, the RFC for TCP shows that this can be done, as it includes a protocol state machine drawn in ASCII art [17].

Extracting the protocol state machine from a prose description can be a lot of work. Typically, the prose will describe constraints on correct sequences of messages, and such constraints are then scattered throughout a long specification document (or several documents). What to do in case of deviations from the happy flow is often left implicit. We once spent several days poring over the RFCs defining SSH (RFCs 4250-4254) to understand the protocol state machine of SSH, which is absent in the RFCs. When doing this we realised that anyone implementing SSH will have to do exactly the same work. Including explicit protocol state machines in a protocol specification, even if these are just partial state machines describing some sub-protocols, can save programmers implementing them a lot of work. This then also helps to ensure more uniform behaviour across different implementations and reduces the room for deviations from the specification, which might introduce security flaws.

[2]Of course, ideally the BNF grammar should be *the* specification, and not just an informative supplement to the specification in English prose.

Beyond the happy flow(s)

The full state machine that has to be implemented is always more complicated than a state machine that just specifies the happy flow: even if all protocol runs conform to a single fixed sequence of messages, the implementation will have to cope with errors, which can be errors in individual messages or errors in the order of messages.

Erroneous messages may be silently ignored by an implementation or may result in the session being aborted – and possibly restarted. In a protocol state machine, ignored messages result in 'self loops', i.e. a transition from a state back to itself. Aborting a session results in many transitions that jump to some error state or back to an initial state. For example, in Fig. 1, which gives the state machine of a payment application on a bank card, we can see many self-loops and many transitions back to the second state from the top, which is effectively the start state of a session.

A robust implementation should be able to handle *any* sequence of inputs. So *for every state* the state machine should specify what should happen *for every possible input*. The technical term for this is that the state machine is *input-enabled*.

Trying to draw all these transitions in a state machine quickly becomes very messy. It is not so clear what the best way is to specify the full state diagram in practice here. Merging transitions which have the same source and destination state, as done in the lower diagrams in Fig. 1, can help in keeping a state machine readable. Omitting the transitions that abort the session or that are ignored from the state machine, and specifying these in prose, can be a practical option. An alternative approach, used in StateCharts [18], is to use nested states, so that common error transitions from a collection of states only have to be drawn once.

V. How things can go wrong

Incorrectly implementing the protocol state machine can result in incompatibilities between different implementations, but also in exploitable security flaws. A flaw may enable Denial-of-Service attacks, namely if a strange sequence of inputs crashes an implementation. Accepting an

incorrect sequence of messages in a cryptographic security protocol can easily break security guarantees the protocol is meant to provide. More generally, a bug in the protocol state machine may allow an attacker to by-pass some security check. And if a protocol consists of various sub-protocols, which in practice is often the case, messages coming in the wrong order can cause unwanted *feature interaction* between the sub-protocols.

One extreme example of an insecure implementation of SSH we came across was an SSH client that did not include any implementation of a protocol state machine whatsoever [8]. This meant that the user could for instance be asked for a username and password before any session key had been established. The programmers had carefully followed the specs in implementing the handling of individual messages, but had completely forgotten to check if these messages came in the correct order. Obviously, the implementation worked fine with any compliant SSH server; that there was additional behaviour that an attacker could exploit would not be noticed in normal use.

We discovered another security vulnerability due to a flawed protocol state machine in a token for internet banking used by one of the largest banks in the Netherlands [6]. This flaw was more shocking because it was not in a small open source code project, but in a commercial project for a bank which presumably has been subjected to thorough security reviews, and which left millions of customers with a flawed (and unpatchable) device. The USB-connected device, which contains a smartcard reader, contained a bug that made it possible to by-pass the crucial security check – the user pressing the OK button – with a non-standard sequence of USB commands. So the protocol state machine implemented in the device had one transition that should not be present, as can be seen in Fig. 2.

Differences between implementations of the protocol state machine may also be used for fingerprinting. Error messages triggered by 'incorrect' sequences of inputs often reveal unique characteristics of a particular implementation. For example, analysis of electronic passports from ten different countries revealed that the nationality could be determined from error messages reported in abnormal sessions [3], even though the protocols for electronic passports have been specifically designed not to leak information to an attacker. There is a comment hidden away in one of the documents that make up the official specification of electronic passports [19] about a standard error message that should be reported in case of incorrect inputs. However, this comment is not very clear and has apparently been overlooked in virtually all implementations we looked at.

We conjecture that bugs in the implementation of a protocol's state machine are less likely to produce a programmable 'weird machine' [20] than bugs in the handling of the format of individual messages. (Indeed, we have never encountered any.) Intuitively, bugs in the protocol state machine may allow an attacker to skip some security-critical step; bugs in parsing the more expressive language of individual messages are more likely to expose a lot of variety in behaviour which an attacker can try to 'program' with carefully crafted inputs.

VI. Model-based testing or stateful fuzzing

To detect flaws in implementations, a formally specified protocol state machine can be used for *model-based testing*, where random sequences of messages are fired at an implementation under test to check if the responses match the ones predicted by the state machine. We have for instance used model-based testing to check the compliance of electronic passports [4], using protocol state diagrams [5] made on the basis of the United Nations ICAO specifications.

Model-based testing against a state machine model is essentially a stateful form of *fuzzing*. Such *stateful fuzzing* can be considered as a next stage after so-called *protocol fuzzing*, where one fuzzes the various fields in the protocol message format. Stateful fuzzing is supported by fuzzing tools such as Peach [21] and SNOOZE [22]. State-based fuzzing has for example been used on the Session Initiation Protocol (SIP) used in VoIP [23], on IEEE 802.11 wireless networks [24], and to a limited extent on GSM phones [25].

Of course, one might hope to avoid the whole problem of having to look for flaws in the implementation of the protocol state machine by generating code from some formal specification. However, as discussed in Section III, generating implementations from specifications seems harder to do for the session language than for the message language. Of course, it is possible to generate code from state machines, and various development methods provide support for this (e.g. UML), but either the models have to be very expressive, or what is generated is a code skeleton that still needs to be refined manually.

VII. Extracting specifications from implementations

Interestingly, it is possible to automatically infer a protocol state machine from an implementation, using just black box testing. The techniques for this date back to work in automata theory, in particular Angluin's L* algorithm [10]. We only came across this technique recently, but it has been used for security analyses by others before, e.g. to analyse botnets [26] and more recently web applications [27].

All that is needed to infer a state machine is a test harness which can fire typical protocol messages at an implementation and record the resulting response. The messages that the test harness can send should cover the different types of messages that occur in the protocol. With such a test harness one can then infer the protocol machine using tools such as LearnLib [11] or Tomte [28].

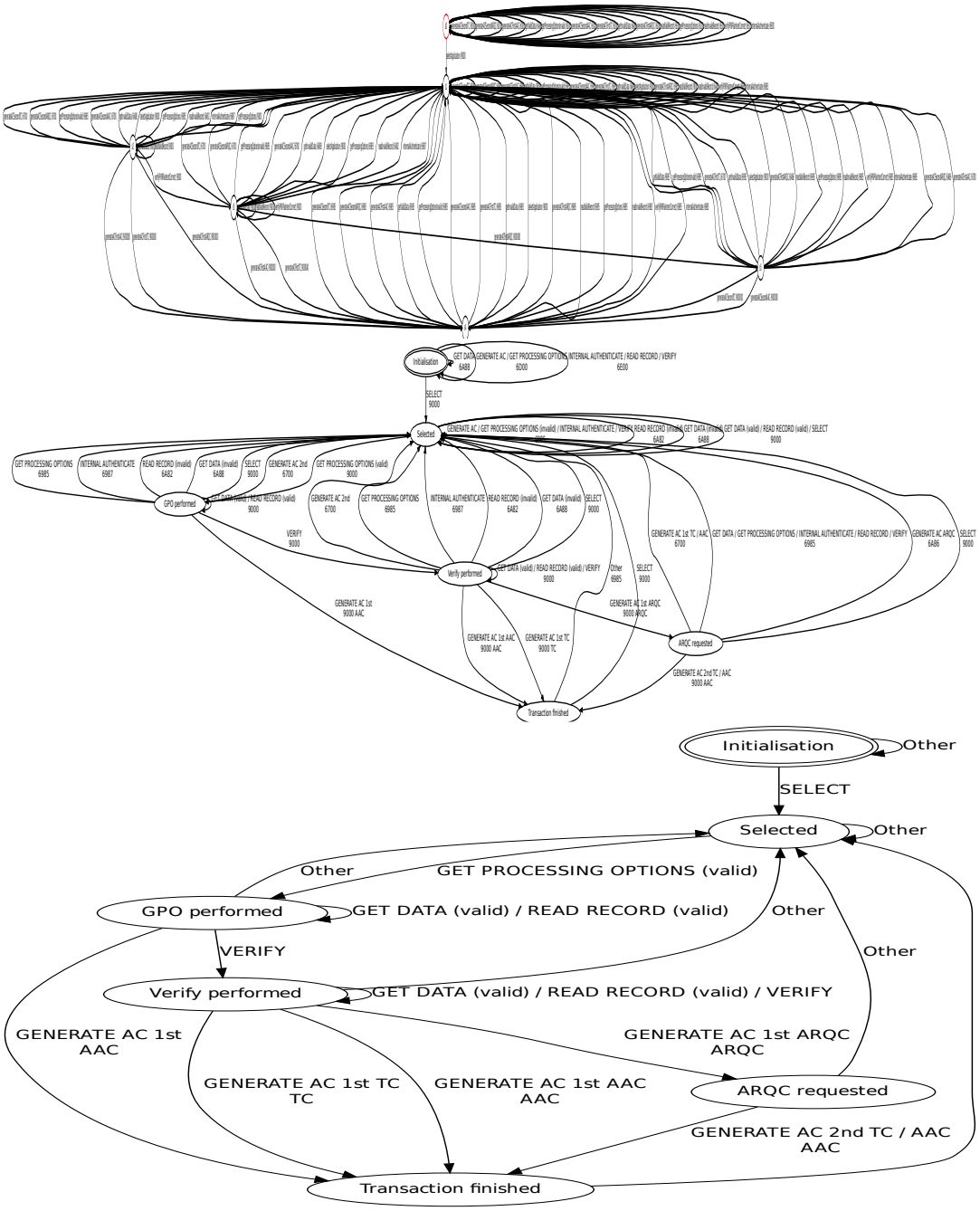

Figure 1. Automatically inferred protocol state machine of a smartcard application in a bank card [1]. The top image shows the raw results, in the middle image transitions with the same source and destination have been merged if they result in the same response, and in the bottom image all transitions with the same source and destination have been merged in one arrow labelled 'Other'.
In the bottom image it is easy to check that verification of the PIN, which happens in the VERIFY branch, must be taken before performing the security critical step, which in this protocol is the branch labelled "GENERATE AC 1st TC".

We first used this technique to extract protocol state machines from applications on the chips in bank cards [1], which nearly always implement a variant of the EMV (Europay-Mastercard-Visa) protocol. Fig. 1 gives an example. Earlier, this technique had already been used on electronic passports [29]. In the bank cards we found a surprising variety in implementations, but no exploitable security flaws.

In some cases making a test harness is very simple. For example, our test harness for EMV bank cards only contains

300 lines of code. For more complex protocols, such as TLS, making a test harness is considerably more work. Still, for a given protocol such a test harness only has to be made once, and it can then be used to analyse any implementation of the protocol. This makes it worthwhile to produce such test harnesses for important standard protocols.

The state machine models obtained in this way are only guaranteed to be an abstraction of the implementation. Without looking at the code of the implementation it is impossible to exclude the possibility that there is additional behaviour that the automated inference did not detect. So a well-hidden backdoor (or Easter egg) in the implementation will not be detected. Still, in our experience many flaws in the program logic typically will be revealed.

Using a Lego robot to operate the keyboard, we also used state machine learning to analyse the flawed internet banking token discussed earlier. The models obtained, shown in Fig. 2, reveal the security flaw in the original device and confirm that this flaw has been fixed in the newer version [7]. The models in Fig. 2 use a limited alphabet of input messages, namely combined messages that occur together in a normal session to perform an online payment. Inferring the state machine of the device with a larger input alphabet reveals a very complex state machine, shown in Fig. 3. We see no reason for it to be so complex: the device only has to be able to (i) ask for a PIN (which is then sent to the smartcard), (ii) display some data, (iii) ask the user to press OK or CANCEL, and then (iv) get the smartcard to sign a transaction if – and only if – the user pressed OK. The complexity is rather worrying, as it complicates the job of ensuring that all the flows are secure. As the device is closed source and implements a proprietary, secret protocol, we can only guess at the causes of this complexity, and whether this is deliberate, due to a sloppy specification of the protocol, or due to a sloppy implementation.

Looking at networking protocols, Fiterau-Brostean at al. used state machine inference on TCP implementations [30]. This revealed differences between the state machines implemented for TCP on Windows 8 and Ubuntu Linux that can be used for fingerprinting [30]. In fact, the differences found are similar to those used by tools such as nmap [31] for OS fingerprinting.

Using the technique on eight TLS implementations [9] revealed a surprising variety in the protocol state machines: all implement a different state machine, as shown in Fig. 4. Most implementations have more states than expected and have behaviour that seems unnecessary. Given that security protocols are notoriously fragile, any superfluous behaviour in an implementation merits serious attention. Indeed, for three of the eight implementations, namely GnuTLS, Java Secure Socket Extension, and OpenSSL, the spurious behaviour raises new security concerns, as discussed in [9]. Additionally, the state diagram inferred for OpenSSL also revealed a security vulnerability already discovered earlier, namely CVE-2014-0224 [32], which has been present since the first release of OpenSSL.

The flaw in Java Secure Socket Extension (caused by the dashed arrow in Fig. 4) has been assigned CVE identifier CVE-2014-6593. It has independently been found by Beurdouche et al. [33], who nicknamed it FREAK; they also recognise the role of flawed state machines as root cause.

In some cases the spurious behaviour found in the TLS implementations is not enough to lead to an actual exploit, but does weaken security guarantees, and so it should be removed. For example, the TLS handshake phase, which establishes the session keys, ends with an integrity check where the client and server exchange and compare MACs (Message Authentication Codes) computed over all the messages that were exchanged as part of the handshake. This guarantees that they saw the same messages. However, in GnuTLS, a HeartbeatRequest message sent *during* the TLS handshake phase will corrupt this MAC, as it has the unwanted side-effect of clearing the buffer used to store the handshake messages. An attacker might use this to by-pass the integrity check. Though in itself this does not provide an exploit, it is an unwanted and unnecessary weakness.

Of course, even for the implementations where the spurious behaviour turned out to be harmless, the extra behaviour simply should not be there, if only to avoid the extra work in assessing the security impact. All this does raise the question what the 'best' – or, the 'correct' – protocol state machine for TLS is. The simplest state machines in Fig. 4 are obvious candidates.

VIII. Conclusions

An important message of the work on language-theoretic security is that input languages should be more precisely defined, in order to tackle a root cause of security flaws in software. We argue that this not only applies to the languages of individual input messages, but also to languages of protocol sessions, i.e. languages of *sequences* of messages. These session languages are typically (and often poorly) described in prose, with sentences describing constraints on message sequences scattered throughout long specifications (or even across multiple documents that make up the specification), and not with a clear and complete protocol state machine. We have seen that several security flaws have their root cause in the flawed implementation of a protocol state machine. How large and important this category of security flaws is remains to be seen. The differences in implemented protocol state machines can also be used for fingerprinting.

The results obtained using state machine inference suggest this is an interesting technique to automatically extract protocol state machines from implementations using black box testing. This can be a useful first step to look for strange behaviour in an implementation. Moreover, the models obtained this way can provide a first step towards clearer

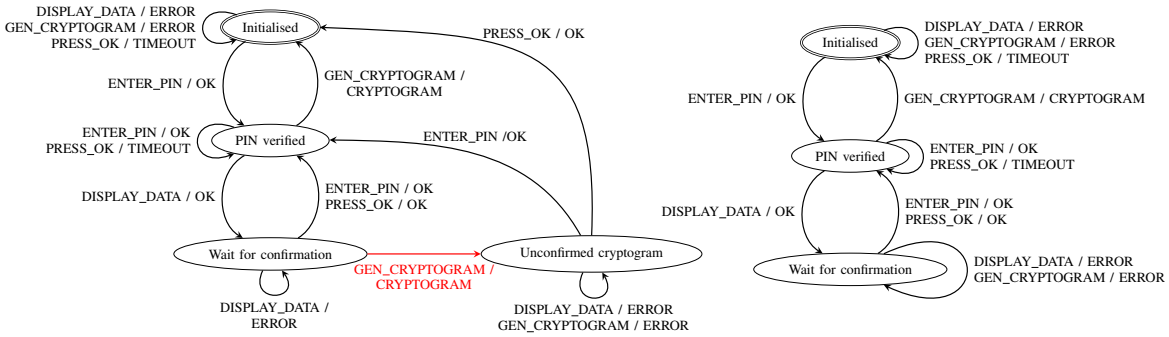

Figure 2. Protocol state machines of the flawed internet banking token (left) and the fixed version (right), for a restricted input alphabet [7]. In the fixed version asking a so-called cryptogram in the bottom state results in an error, and not a cryptogram, because the user has not pressed OK yet.

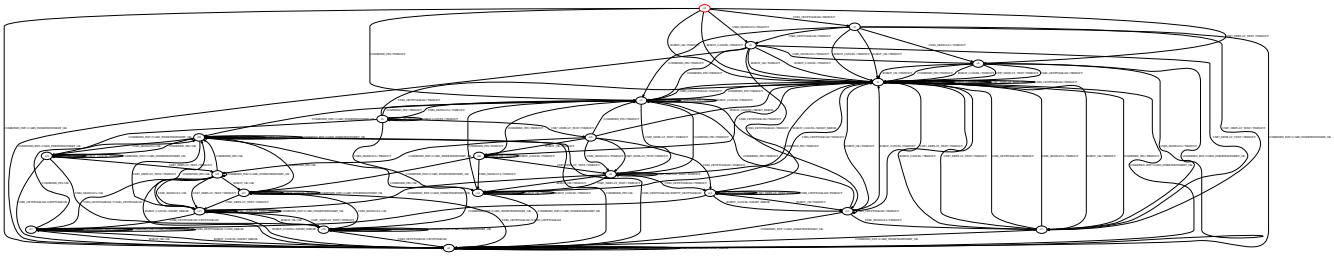

Figure 3. Protocol state machine of the flawed internet banking token, for a more extensive input alphabet. This figure is not meant to be readable, but is included to illustrate the worrying – or, from the attacker's point of view, promising – complexity.

specifications of existing session languages. One obvious direction for future work is applying state machine inference to more protocols, to see how useful this technique is in revealing strange behaviour or security weaknesses. Another question is if we can come up with reference state machine models of important protocols, such as TLS.

Any spurious or non-standard behaviour in a protocol implementation can lead to insecurity, and even if it does not, it makes it harder to check that the implementation is secure. As has been noted many times before [34], [35], it may be time to deprecate Postel's Law – 'Be conservative in what you send, be liberal in what you accept' –, which was introduced in times when security was less of a concern, and also be conservative in what an implementation accepts.

The fact that these session languages are so poorly described in typical specifications is all the more disappointing because there is such a nice specification formalism that can be used for this, namely finite state machines. To summarise our message in a slogan:

> *No more prose specifications of protocol state machines!*

If the full state machine is large and complex, separate state machines of sub-protocols can be given. Introducing names for protocol states, or protocol phases that correspond with sets of protocol states, can be useful for talking about the state machines, and these names can even be used as constants in program code.

Drawing the complete protocol state machine, that is input-enabled and describes all errors that can occur, may result in too many transitions for the result to be conveniently readable. Resorting to English prose to describe the non-happy flows may be the best (or only) option, but this should then not be done in sentences scattered in various places in long documents, but all in one place.

Our original motivation for looking at protocol state machines was to formally specify and verify code [8], [5] or to generate provably correct code [2]. However, in the end we expect that they are more useful simply as a convenient specification formalism: it can help designers of protocols to clearly specify these (and encourage them to keep their protocols simple!), help programmers in producing correct and secure implementations, and help in the security analysis of implementations.

Acknowledgements

The research direction presented in this paper builds on a lot a joint work and discussion with colleagues over the years, in particular, in roughly chronological order: Martijn Oostdijk, Engelbert Hubbers, Wojciech Mostowski, Julien Schmalz, Jan Tretmans, Frits Vaandrager, and Fides Aarts. We also thank Julien Vanegue and Sergey Bratus for their feedback on earlier versions of this paper.

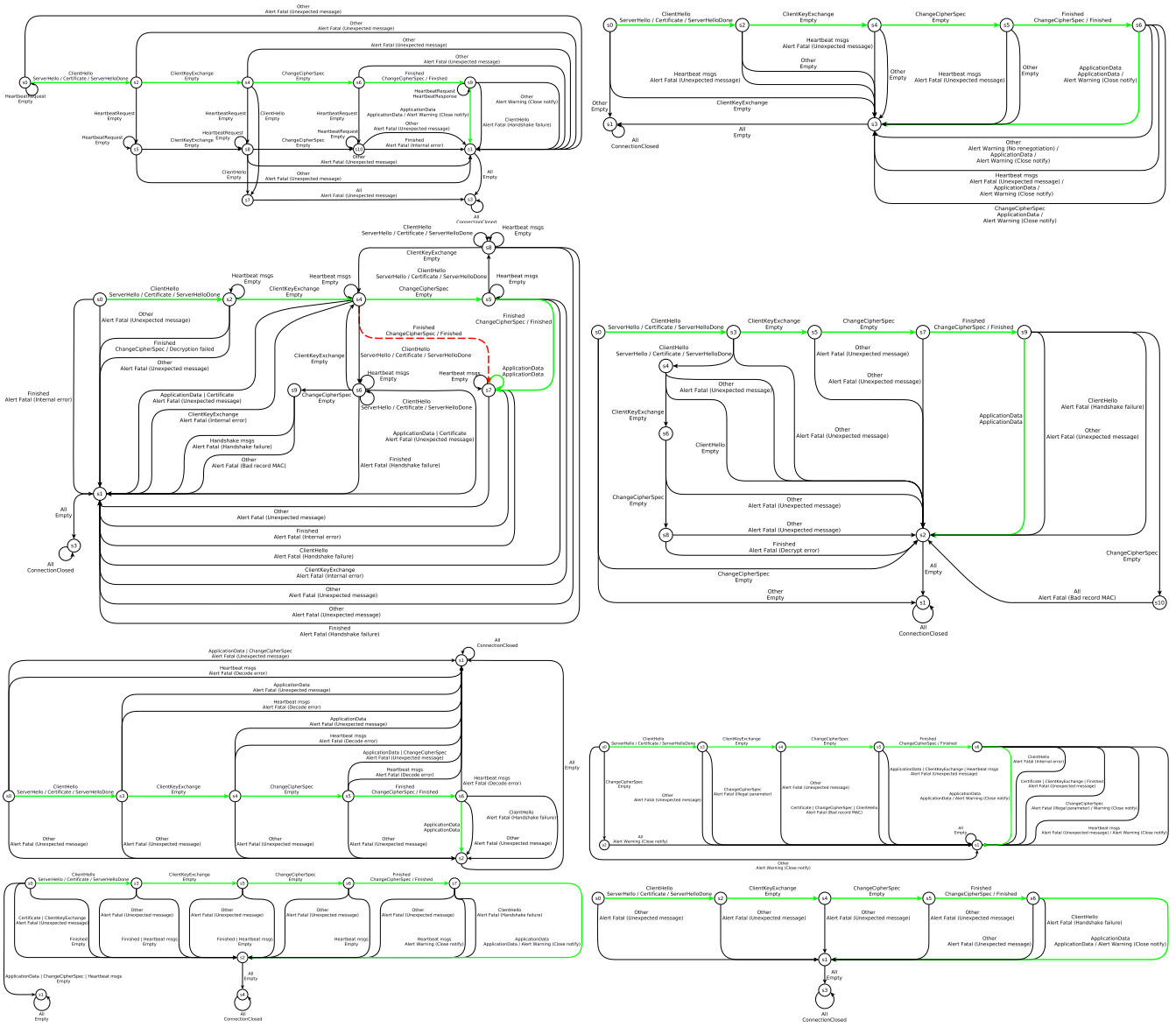

Figure 4. Protocol state machines inferred for eight TLS server implementations. (From top left to bottom right: GnuTLS v3.3.8, PolarSSL v1.3.8, Java Secure Socket Extension v1.8.0, OpenSSL v1.0.1, MiTLS v0.1.3, RSA BSafe C v4.0.4, NSS v3.17.1, and RSA BSafe Java v6.1.1.) The main point here is to show the large variety; for a thorough discussion of the differences and their impact on security see [9].

REFERENCES

[1] F. Aarts, E. Poll, and J. de Ruiter, "Formal models of bank cards for free," in *Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2013, pp. 461 – 468.

[2] E. Hubbers, M. Oostdijk, and E. Poll, "From finite state machines to provably correct Java Card applets," in *Proceedings of the 18th IFIP Information Security Conference*. Kluwer Academic Publishers, 2003, pp. 465–470.

[3] H. Richter, W. Mostowski, and E. Poll, "Fingerprinting passports," in *NLUUG Spring Conference on Security*, 2008, pp. 21–30.

[4] W. Mostowski, E. Poll, J. Schmaltz, J. Jan Tretmans, and R. Wichers Schreur, "Model-based testing of electronic passports," in *Formal Methods for Industrial Critical Systems (FMICS 2009)*, ser. LNCS, vol. 5825. Springer, 2009, pp. 207–209.

[5] W. Mostowski and E. Poll, "Midlet navigation graphs in JML," in *13th Brazilian Symposium on Formal Methods (SBMF)*, ser. LNCS, vol. 6527. Springer, 2010, pp. 17–32.

[6] A. Blom, G. de Koning Gans, E. Poll, J. de Ruiter, and R. Verdult, "Designed to fail: A USB-connected reader for online banking," in *NordSec*, ser. LNCS, vol. 7616. Springer, 2012, pp. 1–16.

[7] G. Chalupar, S. Peherstorfer, E. Poll, and J. de Ruiter, "Automated reverse engineering using Lego," in *8th Usenix Workshop on Offensive Technologies (WOOT 2014)*. Usenix, 2014.

[8] E. Poll and A. Schubert, "Verifying an implementation of SSH," in *WITS'07*, 2007, pp. 164–177.

[9] J. de Ruiter, "Lessons learned in the analysis of the EMV and TLS security protocols," Ph.D. dissertation, Radboud University Nijmegen, 2015.

[10] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, 1987.

[11] H. Raffelt, B. Steffen, and T. Berg, "LearnLib: a library for automata learning and experimentation," in *Formal methods for industrial critical systems (FMICS'05)*. ACM, 2005, pp. 62–71.

[12] G. V. Bochmann, D. Rayner, and C. H. West, "Some notes on the history of protocol engineering," *Computer Networks*, vol. 54, no. 18, pp. 3197–3209, 2010.

[13] R. Seggelmann, M. Tuexen, and M. Williams, "Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension," Internet Engineering Task Force, RFC 6520, 2012.

[14] F. B. Schneider, "Enforceable security policies," *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 1, pp. 30–50, 2000.

[15] S. Bratus, M. L. Patterson, and D. Hirsch, "From shotgun parsers to more secure stacks," *Shmoocon, Nov*, 2013.

[16] E. Poll and A. Schubert, "Rigorous specifications of the SSH Transport Layer," Radboud University Nijmegen, Tech. Rep. ICIS–R11004, 2011.

[17] J. Postel, "Transmission Control Protocol," Internet Engineering Task Force, RFC 793, 1981.

[18] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[19] "PKI for machine readable travel documents offering ICC read-only access, version 1.1," United Nations International Civil Aviation Organization (ICAO), Tech. Rep., 2004.

[20] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina, "Exploit programming: From buffer overflows to weird machines and theory of computation," *Usenix ;login:*, vol. 36, no. 6, pp. 13–21, 2011.

[21] "Peach fuzzing platform," http://peachfuzzer.com.

[22] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "SNOOZE: toward a stateful network protocol fuzzer," *Information Security*, pp. 343–358, 2006.

[23] H. J. Abdelnur, R. State, and O. Festor, "KiF: a stateful SIP fuzzer," in *Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications*. ACM, 2007, pp. 47–56.

[24] S. Keil and C. Kolbitsch, "Stateful fuzzing of wireless device drivers in an emulated environment," 2007, presented at Black Hat Japan 2007. Tool and white paper available at http://www.iseclab.org/projects/vifuzz/.

[25] F. van den Broek, B. Hond, and A. Cedillo Torres, "Security Testing of GSM Implementations," in *Engineering Secure Software and Systems (ESSOS)*, ser. LNCS, vol. 8364. Springer, 2014, pp. 179–195.

[26] C. Y. Cho, D. Babić, E. C. R. Shin, and D. Song, "Inference and analysis of formal models of botnet command and control protocols," in *Proceedings of the 17th ACM conference on Computer and Communications Security (CCS)*. ACM, 2010, pp. 426–439.

[27] M. Buchler, K. Hossen, P. F. Mihancea, M. Minea, R. Groz, and C. Oriat, "Model inference and security testing in the SPaCIoS project," in *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*. IEEE, 2014, pp. 411–414.

[28] F. Aarts, H. Kuppens, J. Tretmans, F. Vaandrager, and S. Verwer, "Improving active Mealy machine learning for protocol conformance testing," *Machine Learning*, vol. 96, no. 1-2, pp. 189–224, 2013.

[29] F. Aarts, J. Schmaltz, and F. Vaandrager, "Inference and abstraction of the biometric passport," in *International symposium on leveraging applications of formal methods, verification, and validation (ISoLa'10)*, 2010, pp. 673–686.

[30] P. Fiterau-Brostean, R. Janssen, and F. W. Vaandrager, "Learning fragments of the TCP network protocol," in *Proceedings 19th Workshop on Formal Methods for Industrial Critical Systems (FMICS 2014)*, ser. LNCS, vol. 8718. Springer, 2014, pp. 78–93.

[31] G. F. Lyon, *Nmap Network Scanning*, 2009. [Online]. Available: https://nmap.org/book/osdetect.html

[32] M. Kikuchi, "OpenSSL #ccsinjection vulnerability (CVE-2014-0224)," 2014, http://ccsinjection.lepidum.co.jp, accessed on August 26th 2014.

[33] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue, "A messy state of the union: Taming the composite state machines of TLS," in *IEEE Symposium on Security and Privacy*. IEEE, 2015, to appear.

[34] D. Geer, "Vulnerable compliance," *Usenix ;login:*, vol. 35, no. 6, pp. 10–12, 2010.

[35] L. Sassaman, M. L. Patterson, and S. Bratus, "A patch for Postel's robustness principle," *Security & Privacy, IEEE*, vol. 10, no. 2, pp. 87–91, 2012.