A Tale of the OpenSSL State Machine: A Large-Scale Black-Box Analysis

Joeri de Ruiter(✉)

Institute for Computing and Information Sciences, Radboud University, Nijmegen, The Netherlands
joeri@cs.ru.nl

Abstract. State machine inference is a powerful black-box analysis technique that can be used to learn a state machine implemented in a system, i.e. by only exchanging valid messages with the implementation a state machine can be extracted. In this paper we perform a large scale analysis of the state machines as implemented over the last 14 years in OpenSSL, one of the most widely used implementations of TLS, and in LibreSSL, a fork of OpenSSL. By automating the learning process, the state machines were learned for 145 different versions of both the server-side and the client-side. For the server-side this resulted in 15 unique state machines for OpenSSL and 2 for LibreSSL. For the client-side, 9 unique state machines were learned for OpenSSL and one for LibreSSL. Analysing these state machines provides an interesting insight in the evolution of the state machine of OpenSSL. Security vulnerabilities and other bugs related to their implementation can be observed, together with the point at which these are fixed. We argue that these problems could have been detected and fixed earlier if the developers would have had the tools available to analyse the implemented state machines.

1 Introduction

TLS (Transport Layer Security) is one of the most widely used security protocols and is used to secure network communications, for example, when browsing the Internet using HTTPS or using email with SMTPS or IMAPS. TLS is the successor of SSL (Secure Socket Layer), originally developed at Netscape. As the name SSL is so widespread, it is often used interchangeably with TLS. The first version of SSL was never released and the second version contained numerous security issues [23]. The third version of SSL was also not without security issues, and these were fixed in the first TLS version [7]. Two more TLS versions were released after this and the fourth one, TLS version 1.3, is currently under development [8,9]. Despite the fact that TLS 1.0 was released in 1999, many servers on the internet today still support SSLv3 and even SSLv2 [2].

Due to its widespread use, the TLS protocol has been the subject of many research projects. For example, it has been analysed using various different formal methods [6,10–14,16,18,19]. These formal analyses focus on the protocol

© Springer International Publishing AG 2016
B.B. Brumley and J. Röning (Eds.): NordSec 2016, LNCS 10014, pp. 169–184, 2016.
DOI: 10.1007/978-3-319-47560-8_11

specifications, while many mistakes are also made in the actual implementation [15]. To counter this, a formally verified TLS implementation has been proposed by combining a formal analysis with an actual implementation [4].

A large proportion of the applications that use TLS to secure their connections use the implementation provided by OpenSSL.[1] The first official release of the OpenSSL project was version 0.9.1c in December 1998, and builds on the code of SSLeay by Young and Hudson. Various forks of OpenSSL exist, such as BoringSSL[2] and LibreSSL[3], which were mainly started with the goal of cleaning up the code and improving its security. Over the years OpenSSL has been plagued with numerous implementation bugs, with sometimes a high security impact. The most well-known example of this is probably the infamous Heartbleed bug.[4]

In this paper we focus on the implementation of state machines of TLS. Every implementation of a protocol needs to implement the corresponding state machine that determines how all the possible messages are handled in different states of the protocol. In [3,22] the state machines of TLS implementations have been analysed, where for various implementations only recent versions were analysed. In [22] a technique called *state machine inference* was used to extract the state machine from TLS implementations by only interacting with it using valid protocol messages. In this paper we will show how we automated the process of using state machine inference in order to analyse a large number of TLS implementations. We will use this to show how the state machine as implemented in OpenSSL changed over the years and what issues could have been prevented should this technique have been available to the developers. In order to do this we learned the state machine for both the client- and server-side of 145 versions of OpenSSL and LibreSSL. We checked BoringSSL as well, but as it does not seem to use version numbering and it is not really intended for use outside of Google we did not perform a large scale analysis of it. We reported our findings regarding several smaller issues related to the state machine implementation in BoringSSL.

2 TLS

In this section we will provide a short introduction to TLS, necessary to understand the results later on. The goal of TLS is to set up an authenticated confidential channel between two parties. The authentication can be mutual, but in most cases it is only the server that is authenticated to the client.

The protocol starts with a handshake that is used to establish the used parameters, including the cipher suite (a combination of a key exchange, encryption and MAC algorithm), perform the desired authentication and establish shared

[1] https://www.openssl.org/.
[2] https://boringssl.googlesource.com/boringssl/.
[3] http://www.libressl.org/.
[4] http://heartbleed.com/.

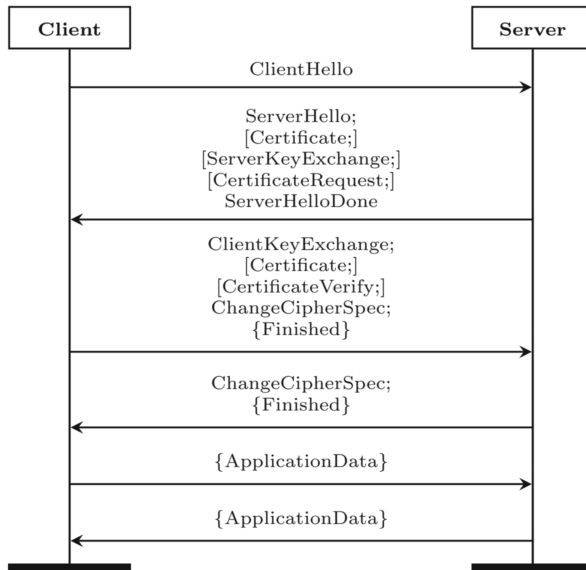

Fig. 1. A regular TLS session. An encrypted message m is denoted as {m}. If message m is optional, this is indicated by [m].

session keys. Different session keys are used for both directions of the communication and for encryption and the computation of the MACs. Once the keys are established, application data can be exchanged, which will be encrypted and authenticated using MACs. In Fig. 1 we provide an overview of a regular TLS session.

To start the handshake, usually the client will send a *ClientHello* message, containing a list of supported ciphersuites and optional extensions. The server will select a ciphersuite and return a *ServerHello* message, as well as other optional messages such as its *Certificate* (used to authenticate the server), the *ServerKeyExchange* message (used in some key exchange algorithms), and a *CertificateRequest* (used to request authentication from the client). The server then indicates it is done by sending a *ServerHelloDone* message. Upon receiving this last message, the client performs the local computations for the key establishment and sends the necessary information to the server in a *ClientKeyExchange* message. If requested by the server, the client also sends the optional *Certificate* and *CertificateVerify* messages to authenticate itself. After this, the client is ready to start encrypting its messages, and it indicates that it will encrypt all following messages by sending the *ChangeCipherSpec* message to the server. This is followed by the *Finished* message, the first encrypted message which is used to provide integrity to the handshake. The *Finished* message contains a keyed hash over all the previous messages that the client sent and received. If this hash does not match the value as expected by the server or the server cannot decrypt the *Finished* message, this can be an indication of a man-in-the-middle attack and the

connection should be closed. If the hash does match, the server will respond by sending the *ChangeCipherSpec* message to indicate it will also encrypt all subsequent messages. This is again followed by an encrypted *Finished* message containing a keyed hash over all previous messages. Once the client accepts the *Finished* message, the client and server are ready to start exchanging data securely using *ApplicationData* messages.

To indicate possible errors during the connection the TLS specification includes *Alert* messages. These alerts can have either a warning or fatal level, where the first kind is only to inform the other party, while the second indicates the protocol should be aborted and the connection closed. The *Alert* messages always include a pre-defined reason, which can be, for example, *Unexpected message*, *Bad record MAC* or *Close notify*. These *Alert* messages can be useful in our analysis as they can indicate interesting conditions. For example, if we receive a *Bad record MAC* alert this can be an indication that the keys on the client and server differ, which is worth looking into in more detail.

3 State Machine Inference

To extract the model of a state machines for a protocol from an implementation, a technique known as *state machine inference* can be used. This technique tries to learn the state machine by only sending protocol messages and observing the responses. This makes it a very useful technique for black-box analysis.

As representation of state machines we use Mealy machines. This gives us a non-ambiguous formal way to describe the learned state machines. A Mealy machine consists of a set of states, of which one is the initial state. Additionally, an input alphabet is specified that describes which messages the system accepts as input. Similarly an output alphabet contains the messages that the system can send as responses. For every state, two functions are defined that map every input message to a corresponding output and to a next state respectively.

In state machine inference, two types of algorithms are used. First, a *learning algorithm* is used to come up with a hypothesis for the implemented state machine. To do this it can send protocol messages to the *system under test* (SUT) and receive the corresponding responses. Which messages can be sent is specified in the input alphabet. The algorithm also has the ability to reset the SUT to its initial state. Once the learning algorithm comes up with a hypothesis for the state machine by exchanging messages and resetting the SUT, this hypothesis is passed on to the *equivalence algorithm*. This algorithm determines whether the hypothesis matches the actual state machine. If this is not the case, the equivalence algorithm returns a counter-example. The counter-example is then fed into the learning algorithm, which uses this to update its hypothesis and continues learning until it comes up with another hypothesis. This process is repeated until the equivalence algorithm accepts a hypothesis. As in practice the equivalence algorithm will not know the actual state machine, the equivalence check will need to be approximated. In the next section we will discuss the concrete algorithms we used for the learning and equivalence algorithm.

4 Setup

For the learning of the TLS state machines we use the tool introduced in [22], which makes use of LearnLib [20]. For the learning algorithm we use of Niese's modification of Angluin's well-known L* algorithm [1,17]. The equivalence checking is done using Chow's W-method [5]. Given an upper bound on the number of states, this algorithm is guaranteed to determine correctly whether the correct state machine is found. As this algorithm can be computationally expensive due to the many messages that are sent, we make use of the improvement to the algorithm previously introduced in our tool. This modification makes use of the fact that if a socket is closed, we know that no more messages will be received. Therefore, queries that have a prefix for which we already know the connection will be closed are not performed, thus significantly reducing the number of queries that are send to the implementation. A nice side-effect of this modification is that if there are no loops in the state machine except for one or more sink states where all messages go to the same state with a ConnectionClosed output, we even have the guarantee that we found the correct state machine without having to know an upper bound on the number of states.

In order to get useful results, the abstract input messages, as used by the learning and equivalence algorithms, need to be converted to correctly formatted TLS messages and reversely, the received responses need to be converted to the abstract output messages before they can be used by the algorithms. This translation is done by the *test harness*, which is basically an (almost) stateless implementation of the TLS protocol. In order to successfully finish a TLS handshake, the test harness keeps track of some minimal notion of state by storing essential data such as, for example, the data used in the key exchange.

As input alphabet for our analysis, we made use of a minimal set of TLS messages, that are necessary to establish a successful connection. To test the server-side these are: ClientHello, ClientKeyExchange, an empty client Certificate, ChangeCipherSpec, Finished and two ApplicationData messages, one with a HTTP GET request and one without any data. When sending a ClientHello message, we reset the buffer used to collect all the exchanged messages that need to be hashed for the *Finished* messages. For the client-side testing we use the following messages: ServerHello, Certificate, an empty Certificate, ServerHelloDone, ChangeCipherSpec, Finished and again the same two ApplicationData messages as before.

To be able to learn the state machine for as many implementations as possible we completely automated the learning process. First, a crawler is used to download all versions available on the website or FTP server of a specific implementation. After this the downloaded sources are automatically extracted and compiled. This process is implementation specific and might require some tweaks to be able to build older versions. However, once the required steps are known the building of all downloaded versions is automated. Once we have a binary executable, we perform a sanity check in order to see whether the implementation is able to set up a valid TLS connection with our TLS test framework. Some older implementations are not able to do this, for example, as they do

not support TLS yet but only SSLv3 or older. We ignored these older versions in our analysis and focus on only implementations that support at least TLS 1.0. For the versions that do pass the sanity check, the configuration files that are necessary for the actual state machine inference are generated. This is done such that every version uses a unique port so there is no interference between different versions that try to listen on the same port. Once we have the necessary configuration files, the learning is started using our tool from [22]. It is possible to perform the learning in parallel due to the usage of a unique port for each version. Using this process we are able to automatically infer the state machine for many versions of different TLS implementations. All software and models are available online.[5]

5 Analysing the OpenSSL State Machines

Using our automated process we learned the state machine for both the server-side and client-side for 111 different version of OpenSSL and 34 versions of LibreSSL. The latest versions of OpenSSL that we analysed were 1.0.1t, 1.0.2h and 1.1.0-pre4. For LibreSSL the latest version was 2.4.0. For the learning a machine with an Intel Xeon E5-2420 CPU was used. The time required for the learning varied from 3 min for more recent implementations of the server-side to about 2 h for older server-side implementations. These 2 h were exceptional though, and in general the running time per implementation was well below 20 min for both server- and client-side. Below we will discuss our analysis of the state machines we learned for the server- and client-side in Sects. 5.1 and 5.2 respectively.

5.1 Server-Side

For the server-side, the learning process resulted in 15 unique state machines for OpenSSL. The learning of the LibreSSL server-side state machine resulted in only two different state machines. One of these state machines was equal to one of the OpenSSL state machines. In Fig. 2 an overview is given of the different state machines for the server-side and their overlap between different branches of OpenSSL. In this figure we excluded the various beta versions that we learned, leaving 12 different state machines.

The oldest version of OpenSSL for which we learned a state machine is version 0.9.7, released in December 2002. For the server-side this state machines contains 17 states (see Fig. 3), the highest number for all state machines we learned. When analysing the state machine we observe several explanations why the state machine contains so many states. First, it is possible to start a renegotiation after completing a handshake successfully by starting a new handshake. In our case this does not lead to a successful data exchange, and therefore the number of states used for a handshake is already doubled (states 12, 13, 16 and 14).

[5] http://www.cs.ru.nl/~joeri/.

0.9.7 (17)	0.9.7c (14)	0.9.7e (14)						
		0.9.8f	0.9.8l (11)	0.9.8m (10)	0.9.8s (10)	0.9.8u (12)	0.9.8za (9)	
				1.0.0	1.0.0f	1.0.0h	1.0.0m	
						1.0.1 (14)	1.0.1h	1.0.1k (8)
								1.0.2 (7)
								1.1.0-pre1 (6)

Fig. 2. Overview of the 12 state machines of the server-side for different versions of OpenSSL. The version number indicates the first version in a particular branch that a state machine was used. Per unique state machine the number of states is included.

Secondly, the server accepts empty Certificates from the client after a ClientHello message, which adds an additional state for every handshake attempt (i.e. states 5, 8 and 13).

Though these functionalities can still be seen as genuine, we also observe some clearly erroneous behaviour. For example, when after the initial ClientHello immediately a ChangeCipherSpec message is sent, the connection is not closed and the handshake can still be finished by sending a ClientKeyExchange and Finished message (the path through states 1, 6, optionally via 8, to 9 and finishing in 2). The Finished message is not accepted from state 9 however and instead a *Decrypt error* alert is returned. This additional behaviour is the result of a serious security issue that we will discuss in more detail later.

Other observations include the fact that empty ApplicationData messages are always ignored, except if it is the first message that is sent, in which case the connection is closed. Also, it is possible to send the ClientHello message numerous times at the beginning of a handshake as there is a self-loop with this message after the initial ClientHello message in state 1. A possible explanation for this is support for a feature called *Server-Gated Cryptography*. This is a legacy feature that resulted from the export restrictions on cryptography. Under these restriction strong cryptography was still allowed for financial transactions, so if a client asked for a weak export cipher the server could indicate that it was allowed to use the stronger ciphers and the client could send a new ClientHello message containing the stronger ciphersuites.

If after a successfully completed handshake a ChangeCipherSpec message is sent (from states 10, 12 and 13), all subsequent messages are replied to with a *Bad record MAC* alert.

Fig. 3. State machine for the server-side of OpenSSL version 0.9.7.

In version 0.9.7c the state machine is changed and the number of states is reduced to 14. This is due to the fact that the server no longer accepts certificates from the client during the handshake, which results in states 5, 8 and 13 being dropped from the previous state machine. According to the changelog the server now only accepts a certificate if it requested one using a *CertificateRequest* message in order to comply with the official specifications.

The state machine then already changes again in version 0.9.7e. The number of states stays the same though and the only change is that Alert messages are now always sent before the connection is closed after a handshake is initialised. This wasn't the case before for the ChangeCipherSpec and ApplicationData messages (see, for example, state 7 of version 0.9.7). The state machine then stays stable until the end of the branch (version 0.9.7m) and is also the same for the first versions of the 0.9.8 branch that we learned.

Then in November 2009 version 0.9.8l was released, which contains only 11 states. Looking at the learned model, we can see that it is no longer possible to perform a renegotiation after a successfully completed handshake as we previously observed. Around the same time as this release, details are published on a serious vulnerability that is present in many TLS implementations (CVE-2009-3555). This issue made it possible for a man-in-the-middle to inject plaintext data at the beginning of a TLS session. The attacker starts a TLS connection with a server that the victim's client want to speak to. The attacker can then send any data to the server. After this it will start a renegotiation with the server, whereby it forwards the original TLS messages from the victim. The victim does not realise it is performing a renegotiation as it looks the same as the initialisation of a connection. The server will consider it a renegotiation and append the data from the client to the data it initially received from the attacker. The attacker won't be able to eavesdrop on the data that is exchanged between the victim and the server, but by only prepending data it has been shown that, for example, credentials could be stolen.[6] When this issue was reported, developers of different implementations and the IETF came together in "Project Mogul" to find a solution. As the issue is caused by the way renegotiation is performed, OpenSSL initially responded by disabling renegotiation completely, as we observed in the learned state machine.

We also observed some new strange behaviour after the handshake is successfully completed (state 8). Every message, other than ApplicationData or ChangeCipherSpec, is initially ignored, but every following message results in a decryption failure on our side. When analysing the network traffic we noticed that this was due to the fact that the server sent plaintext Alert messages, even though all messages should have been encrypted at this point. We also observe that it is still possible to send a ChangeCipherSpec both directly after the first ClientHello (from state 1) and after a successful handshake (from state 8). As before these paths eventually lead to a *Decrypt error* alert and *Bad record MAC* alert respectively.

---

[6] http://www.securegoose.org/2009/11/tls-renegotiation-vulnerability-cve.html.

In February 2010, RFC 5746 was released [21]. This RFC specifies a secure way to perform renegotiation. This RFC is implemented in the same month in version 0.9.8m. In the state machine multiple ClientHello messages are accepted again, and the strange behaviour that resulted in a decryption error in our framework is no longer present.

At the beginning of 2012, version 0.9.8s was released. The state machine contains 10 states, but we see that a ClientHello is only accepted once now. A second ClientHello is still responded to in the usual way with a ServerHello, Certificate and ServerHelloDone message, though the connection is closed immediately after this. At the end of 2011 it was reported that allowing arbitrary ClientHello messages at the start constitutes a denial-of-service attack as the server has to perform significantly more computations upon receiving a ClientHello than the client (CVE-2011-4619). This issue explains why only one ClientHello is accepted now. However, for Server Gated Cryptography we would still expect to see two ClientHello messages and if a message is rejected it should be responded to with an Alert message and not a valid ServerHello, Certificate and ServerHelloDone message. These issues are fixed in version 0.9.8u, where at most two ClientHello messages are accepted, which increases the number of states again to 12.

Before, we observed the early ChangeCipherSpec, directly after the first ClientHello message, and subsequent messages. This part of the state machine was due to a serious security flaw which was eventually discovered by Kikuchi (CVE-2014-0224).[7] By sending a ChangeCipherSpec message too early, i.e. before the keys have been established, the keys are calculated using an empty master secret and therefore only depend on information known to a possible attacker. Due to the way the Finished message is computed in version 1.0.1, it was vulnerable to decryption of the TLS connection by an attacker who is able to eavesdrop on the connection and even complete hijacking of the connection by a man-in-the-middle. A detailed analysis of this bug is given by Langley on his blog.[8] In version 0.9.8za we see that the ChangeCipherSpec message is no longer accepted directly after the ClientHello message (see Fig. 4). The ChangeCipherSpec is however still accepted directly after a successful handshake.

Branch 1.0.0 completely follows the state machine from branch 0.9.8. Branch 1.0.1 starts with a different state machine though, after which it start using the same state machine as 0.9.8za from version 1.0.1h, to finally end with a different state machine again after 1.0.1k. In version 1.0.1 the early ChangeCipherSpec is accepted as with the earlier versions of 0.9.7 and 0.9.8. However, instead of finishing with a *Decrypt error* from the server-side, our framework cannot decrypt any messages it receives from the server.

Version 1.0.1k (see Fig. 5) was released after we reported the issue regarding the ChangeCipherSpec message following a successfully completed handshake. This behaviour was the result of a bug that resulted in the keys being reset to their initial values and the same key being used for both directions (i.e. from client to server and from server to client). This breaks the protection measures in

[7] http://ccsinjection.lepidum.co.jp/.
[8] https://www.imperialviolet.org/2014/06/05/earlyccs.html.

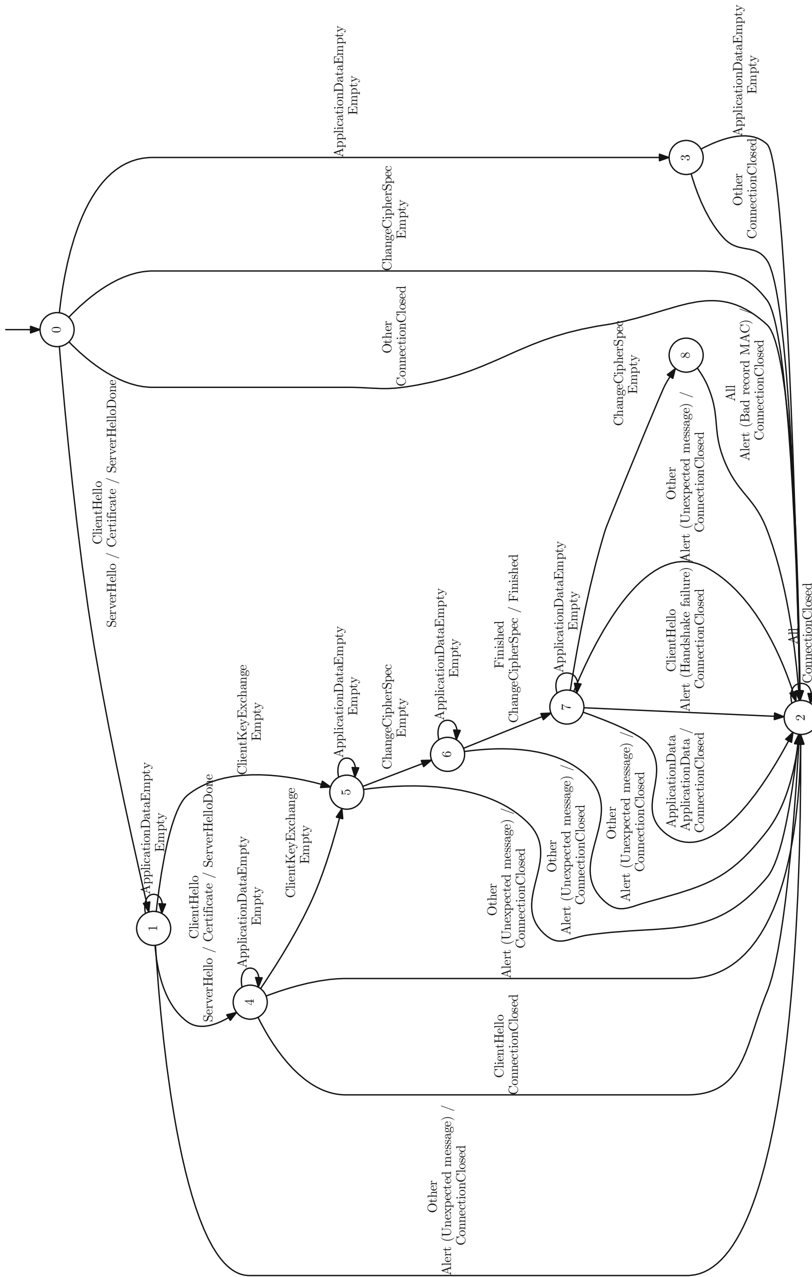

Fig. 4. State machine for the server-side of OpenSSL version 0.9.8za.

Fig. 5. State machine for the server-side of OpenSSL version 1.0.1k

place against replay attacks. In version 1.0.1k we can see the superfluous ChangeCipherSpec message is no longer accepted. At the same time we also checked a development version in which the second ClientHello was replied to with correct messages (ServerHello, Certificate, ServerHelloDone) but immediately after this the connection was closed. This was similar to the behaviour that we observed in version 0.9.8s and was fixed before the code was ever released. From version 1.0.1k the state machine stays stable until the end of the branch.

The state machine for the 1.0.2 branch is stable, except for the beta versions, and contains 7 states. This is one state less than version 1.0.1k, which is due to the fact that Server Gated Cryptography is no longer supported and only one ClientHello is accepted at the beginning of the handshake. In the pre-releases for 1.1.0 the number even drops to 6 states which is caused by the implementation accepting empty ApplicationData messages in every state, even the initial one. In branch 1.1.0, a new implementation is introduced for the state machine.

LibreSSL starts in version 2.0.0 with the same state machine as OpenSSL 0.9.8za. In version 2.2.1 the state machine changes, but the issue we found in OpenSSL with the ChangeCipherSpec message after a successful handshake is still present.

5.2 Client-Side

For the client-side, 9 unique state machines were learned for OpenSSL and one for LibreSSL, which was equal to the latest one from OpenSSL. An overview of the client-side state machines for OpenSSL is given in Fig. 6. Two state machines are again excluded here as they are unique for beta versions.

The first two state machines that we learned did not result in usable state machines. This was due to the fact that these versions did not have extensions enabled by default and therefore rejected our ServerHello messages. In version 0.9.8j, extensions were enabled by default and we get the first model suitable

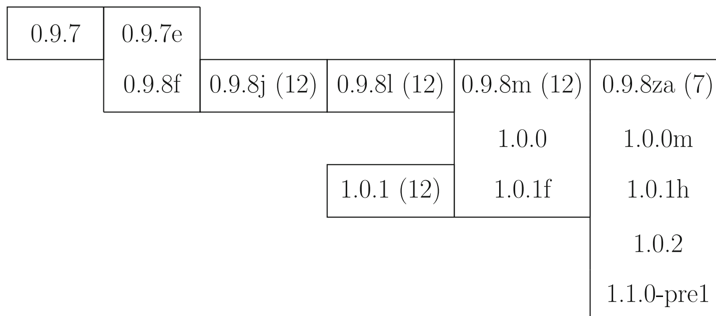

Fig. 6. Overview of the 7 state machines of the client-side for different OpenSSL versions. The version number indicates the first version in a particular branch that a state machine was used. Per unique state machine the number of states is included, except for the first two for which no usable state machine was learned.

for analysis. This state machine contains 12 states and displays some unexpected behaviour. After a successful handshake, most unexpected messages are replied to with a ClientHello message before the connection is closed. It is however possible to send a ChangeCipherSpec message followed by a ServerHello. The ServerHello is responded to with a ClientHello, after which a complete new handshake can be performed. After the ChangeCipherSpec the messages cannot be decrypted any more, indicating that a key is used which is different than expected. The same is the case if a ChangeCipherSpec is sent too early, namely after the ServerHello or the ServerCertificate. This last behaviour matches that of the server-side, which is caused by the vulnerability reported by Kikuchi.

In version 0.9.8l, the number of states stay the same, but the client no longer sends ClientHello messages and renegotiation seems completely disabled. This matches what we observed for the server-side, where renegotiation was disabled to prevent a serious security issue. Renegotiation seems to be re-enabled in version 0.9.8m, as the client sends ClientHello messages again just before closing the connection when receiving an unexpected message after a successful handshake. We are no longer able to perform renegotiations though as our framework does not implement the secure renegotiation as specified in RFC 5746 [21]. From version 0.9.8za, when the bug reported by Kikuchi is fixed, the ChangeCipherSpec message is no longer accepted directly after a successful handshake.

As can be seen in Fig. 6, the next branches implement the same state machines as branch 0.9.8, except for version 1.0.1. This state machine is almost identical to the one from 0.9.8 m though. The only difference is that, upon receiving a Finished message in the initial state, the connection is immediately closed in version 1.0.1, while an alert is sent first in version 0.9.8m. For LibreSSL all versions resulted in the same state machine as OpenSSL version 0.9.8za.

6 Conclusion

By just looking at the state machines, inferred in our automated process, we are able to analyse the evolution of OpenSSL, without requiring an analysis of the source code. Due to our automated process we were able to learn the state machine for 145 different versions for both the server- and client-side. Various bugs can be spotted by only analysing the learned models, and indeed we even reported several new bugs to different developers. Observable bugs can be serious security flaws, such as the one reported by Kikuchi, that have been present for many years, and might have been fixed earlier if only the developers had the tools to analyse their state machine implementation. We can see how the state machine improves over time and how the current versions seems quite clean. Would the developers have had the tools, it might not have taken almost 14 years to get to this state.

Having access to the source code, the developers could possibly leverage this fact in their analysis of the implemented state machine. However, language specific tools would be needed for this and the code might need to be instrumented to be able to use these tools. By using a black-box analysis, as used in this paper, developers can use generic tools that work independent of implementation details.

In future work we intend to extend this analysis to other implementations. Next to this, we plan to add automated analysis to our testing framework in order to make it easy for developers to spot unexpected or strange behaviour and observe changes between versions. We expect this can be a helpful tool as currently developers have many tools to perform, for example, analysis of memory usage or even static analysis of their code, but a tool to check exactly what state machine is implemented is currently lacking.

To conclude, state machine inference is a useful technique when analysing implementations and a large-scale analysis of state machines can tell an interesting tale about the evolution of a protocol implementation.

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987)
2. Aviram, N., Schinzel, S., Somorovsky, J., Heninger, N., Dankel, M., Steube, J., Valenta, L., Adrian, D., Halderman, J.A., Dukhovni, V., Käsper, E., Cohney, S., Engels, S., Paar, C., Shavitt, Y.: DROWN: breaking TLS using SSLv2. In: 25th USENIX Security Symposium (USENIX Security 2016), pp. 689–706. USENIX Association, Austin, August 2016
3. Beurdouche, B., Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.Y., Zinzindohoue, J.K.: A messy state of the union: taming the composite state machines of TLS. In: 2015 IEEE Symposium on Security and Privacy, pp. 535–552 (2015)
4. Bhargavan, K., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.: Implementing TLS with verified cryptographic security. In: 2013 IEEE Symposium on Security and Privacy, pp. 445–459 (2013)
5. Chow, T.: Testing software design modeled by finite-state machines. IEEE Trans. Softw. Eng. **4**(3), 178–187 (1978)
6. Díaz, G., Cuartero, F., Valero, V., Pelayo, F.: Automatic verification of the TLS handshake protocol. In: Proceedings of the 2004 ACM Symposium on Applied Computing, SAC 2004, pp. 789–794. ACM (2004)
7. Dierks, T., Allen, C.: The TLS protocol version 1.0. RFC 2246, Internet Engineering Task Force (1999)
8. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) protocol version 1.1. RFC 4346, Internet Engineering Task Force (2006)
9. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) protocol version 1.2. RFC 5246, Internet Engineering Task Force (2008)
10. Gajek, S., Manulis, M., Pereira, O., Sadeghi, A.-R., Schwenk, J.: Universally composable security analysis of TLS. In: Baek, J., Bao, F., Chen, K., Lai, X. (eds.) ProvSec 2008. LNCS, vol. 5324, pp. 313–327. Springer, Heidelberg (2008). doi:10.1007/978-3-540-88733-1_22
11. He, C., Sundararajan, M., Datta, A., Derek, A., Mitchell, J.C.: A modular correctness proof of IEEE 802.11i and TLS. In: Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, pp. 2–15. ACM (2005)
12. Jager, T., Kohlar, F., Schäge, S., Schwenk, J.: On the security of TLS-DHE in the standard model. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 273–293. Springer, Heidelberg (2012). doi:10.1007/978-3-642-32009-5_17

13. Kamil, A., Lowe, G.: Analysing TLS in the strand spaces model. J. Comput. Secur. **19**(5), 975–1025 (2011)
14. Krawczyk, H., Paterson, K.G., Wee, H.: On the security of the TLS protocol: a systematic analysis. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8042, pp. 429–448. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40041-4_24
15. Meyer, C., Schwenk, J.: SoK: lessons learned from SSL/TLS attacks. In: Kim, Y., Lee, H., Perrig, A. (eds.) WISA 2013. LNCS, vol. 8267, pp. 189–209. Springer, Heidelberg (2014). doi:10.1007/978-3-319-05149-9_12
16. Morrissey, P., Smart, N.P., Warinschi, B.: A modular security analysis of the TLS handshake protocol. In: Pieprzyk, J. (ed.) ASIACRYPT 2008. LNCS, vol. 5350, pp. 55–73. Springer, Heidelberg (2008). doi:10.1007/978-3-540-89255-7_5
17. Niese, O.: An integrated approach to testing complex systems. Ph.D. thesis, Dortmund University (2003)
18. Ogata, K., Futatsugi, K.: Equational approach to formal analysis of TLS. In: 2005 Proceedings of the 25th IEEE International Conference on Distributed Computing Systems, ICDCS 2005, pp. 795–804. IEEE (2005)
19. Paulson, L.C.: Inductive analysis of the internet protocol TLS. ACM Trans. Inf. Syst. Secur. **2**(3), 332–351 (1999)
20. Raffelt, H., Steffen, B., Berg, T.: LearnLib: a library for automata learning and experimentation. In: Formal methods for industrial critical systems (FMICS 2005), pp. 62–71. ACM (2005)
21. Rescorla, E., Ray, M., Dispensa, S., Oskov, N.: Transport Layer Security (TLS) renegotiation indication extension. RFC 5746, Internet Engineering Task Force (2010)
22. de Ruiter, J., Poll, E.: Protocol state fuzzing of TLS implementations. In: 24th USENIX Security Symposium (USENIX Security 2015). USENIX Association, Washington, D.C., August 2015
23. Turner, S., Polk, T.: Prohibiting Secure Sockets Layer (SSL) version 2.0. RFC 6176, Internet Engineering Task Force (2011)