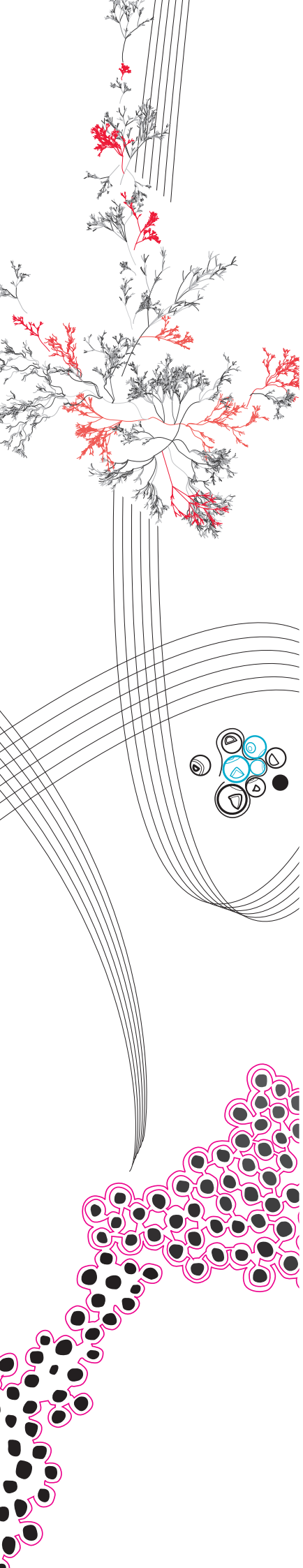
March 29, 2016

ACTIVE STATE MACHINE LEARNING - FINAL PAPER

SOFTWARE TESTING AND REVERSE ENGINEERING

Roy Kokkelkoren (s1617761), Vladyslav Cherednychenko(s1750763)

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Documentnumber
3TU / EIT: Cyber Security Specialization — 201500036



UNIVERSITY OF TWENTE.

Contents

1 Introduction **3**

2 Techniques **3**
2.1 Exploration 3
2.2 Validation 4

3 Tools **5**

4 Applications **6**

5 Results **6**
5.1 Double ClientHello 8
5.2 No ClientHelloDHE 8
5.3 No ClientKeyExchange 8

6 Conclusion **9**

1 Introduction

The following sections contain review of tools and techniques that were used during a research on Active State Machine Learning and results of this research as well.

Aim of this research was to analyze two different TLS[1] implementations: OpenSSL and some other. For the second implementation we've chosen LibreSSL[2] because it's TLS implementation is independent of OpenSSL. Furthermore, we have written a small Python TLS server using the Python TLS library in accordance with the recommendations from official Python documentation.

2 Techniques

Despite impressive progress in automated program analysis there are many obstacles that researchers have to overcome. One of the most popular approaches in this field is called Active Automata Learning[2] which principles we are going to discuss in this report. Generally, program analysis is divided into three categories:

- **Black-box**, when only binary of a program is given
- **White-box**, when the full source code of program is given
- **Grey-box**, when previous two are combined in different ways

While classical program analysis works on the level of a programs state space as constituted by heap contents, local variables etc., active automata learning drastically narrows this perspective, merely focusing on the observable (I/O) behavior. Aim of active automata learning is to infer a finite-state model of a program's (SUL[3]) observable behavior. Interface to the program should be given as a starting point. Active automata learning can also be adapted to grey-box and dynamic white-box analysis methods.

A tool that we have used for our research is the implementation of the Active Automata Learning algorithm. The learning process is divided into two phases:

1. **Exploration phase**, where sequences of input symbols (membership queries) are executed and their outputs are analyzed. The result of this phase is a conjectured hypothesis modeling the SULs behavior.
2. **Validation phase**, where the conjectured hypothesis from the previous state is checked for completeness and correctness with respect to systems behavior. This process realizes the so-called equivalence query.

2.1 Exploration

The exploration is done using LearnLib[4] - a modified version of Angluins L* algorithm[1].

We will briefly run through the L*. So the outputs of membership queries are organized in an observation table which is vertically divided into two parts: lower and upper. Both parts are labeled with words (prefixes) over the input alphabet. Prefixes in the upper part are called short prefixes (S_p), prefixes in the lower part long prefixes (L_p).

S_p serve as unique representatives for the unknown states in the target system. L_p are the one-letter extensions to those in the upper part and correspond to transitions in the hypothesis. Columns in the observation table have the purpose of distinguishing states in the target system

[1] Transport Layer Security
[2] http://www.libressl.org/
[3] System Under Learning
[4] http://learnlib.de/

and are called suffixes (*D*). A cell contains the outcome of executing the concatenation of a prefix (row label) and a suffix (column label) on the system, reduced to the part attributable to the suffix. The contents of a row can be seen as a partial residual finitely approximating the Nerode relation. It is used to distinguish rows in the upper part (states), and match transitions from the lower part with a unique target state.

After the first iteration we have an initial state of the system as a short prefix, a predefined set of suffixes is used to characterize this state. If a transition reveals previously unseen behavior with respect to a certain suffix, a new state has been discovered. Consequently, the corresponding row is moved to the upper part of the tables, and the long prefixes will be augmented by all one-letter extensions of the corresponding prefix. Otherwise, the table is called closed, and we can use it to construct a hypothesis.

The counterexample mentioned before is a word w for which the output of a real system differs from the output predicted by the hypothesis. It gives a rise to at least one additional state in the hypothesis.

2.2 Validation

Obviously, the validation requires full knowledge about the SUT[5], so in practice it is usually approached by conformance testing. If the model is found to be incorrect, the equivalence query yields a counterexample which can be used to refine the hypothesis. Otherwise, the learning is stopped.

For the black-box scenario no executable specification is available, so a method for approximation of equivalence queries is needed. A well-known method for this is a Chow's W-Method[4].

This method consists of 3 major steps:

1. estimate the maximum number of states in the correct design;
2. generate test sequences based on the design (which may have errors);
3. verify the responses to the test sequences generated in step 2;

Estimation. As far as there's no access to the correct design of a system, human judgment should be involved in this process.

Generation of test sequences. The test set required for validation is the concatenation of so-called sequences **P** and **Z**. **P** represents any set of input sequences such that for every transition from state A_i to A_j on input *x* there input sequences *p* and *px* in **P** so that *p* forces the machine into state A_i from its initial state[4]. T. S. Chow recommends to construct **P** by first building "testing tree".

Z is defined to be the following: $W \bigcup X \cdot W \cdots \bigcup X^{m-n} \cdot W$, Where:

- W is a set of input sequences that can distinguish between the behaviors of every pair of states
- X is the input alphabet
- $X^{i+1} = X \cdot X^i$

Verification. According to T. S. Chow, there 2 ways verification can be performed:

1. *Test Mode* - the list of correct responses of the SUT for certain inputs is constructed first, then compared with results of automated testing.
2. *Walk-Through Mode* - input sequences and corresponding responses are checked for correctness by a walk-through procedure based on specification (which is not an easy task because specification is not available).

[5]System Under Test

3 Tools

In the previous section we described techniques that are used to automatically analyze and describe states of a system with unknown specification. In this section we will go deeper into tools that were actually used to conduct this research.

In order to perform the research on the TLS implementations, J. de Ruiter and E. Poll used LearnLib which is a modified version of Angluin's L* algorithm. The LearnLib algorithm requires a list of messages it can send to the SUT[6], which is known as the *input alphabet*. It also requires a command which is sued to reset the SUT to its initial state. By sending numerous messages from the *input alphabet* LearnLib will create a hypotheses for the state machine. This hypotheses is then compared to the actual state machine, if the models are not equivalent the LearnLib will use the difference to refine its hypothesis.

LearnLib does therefore require an actual state machine, however because the tests are performed using a black-box methodology the actual state machine is not known. Therefore in order to compare the models the equivalence check has to be approximated. J. de Ruiter and E. Poll used Chow's W-method to perform this model-based testing which then is used to perform the equivalence check. J. de Ruiter and E. Poll have improved Chow W-method with a specific property of the TLS protocol. This greatly improved the performance required to use Chow's W-method. They used the property of the TLS protocol in which the protocol returns *Connection Closed* to every message send after a connection is closed. Improving the W-method using this property reduced the computation time using the full *input alphabet* from 4:09 hours to 0:27.

As mentioned above, the modification of *L** algorithm called LearnLib was used to execute learning process. Generally, LearnLib is a framework for automata learning. More specifically, it implements an interface for running membership queries and equivalence queries.

LearnLib has following central interfaces:

- LearningAlgorithm, which encapsulates implementations of learning algorithms. It has three methods:
  1. startLearning - start the initial learning round
  2. getHypothesisModel - provides access to the current conjecture in the form of an automaton model
  3. refineHypothesis - provide a counterexample disproving the incomplete conjecture and trigger another learning round
- MembershipOracle which encapsulates any structure that can answer membership queries, i.e., which provides information on the target system's reaction to provided input words
- EquivalenceOracle which will return a counterexample if the equivalence oracle finds a behavioral mismatch. Otherwise the equivalence oracle will attest the adequateness of the provided conjecture by not delivering such a counterexample

From the above we can see that the LearnLib provides us with main tools that are needed for hypothesis construction. For our research we received an improved version of code that was used by J. de Ruiter and E. Poll in their TLS analysis[5]. Improvements of code were done by Mark Janssen[7]. This code will be denoted as TLSTestService.

The best part about the TLSTestService is it's flexibility and ability to adapt to different TLS implementations, so we didn't have to update the code itself, only configuration file required some changes.

There are 2 configuration files in TLSTestService: client and server. Main sections of the configuration file are:

[6] System Under Test

[7] Mark Janssen (mark@ch.tudelft.nl)

- cmd - sets a command line used to execute tests
- alphabet - input alphabet
- learning_algorithm - sets a learning algorithm type

The most interesting part here is the input alphabet. For the client part following inputs were used: *ServerHelloRSA, ServerCertificate, CertificateRequest, ServerHelloDone, ChangeCipherSpec, Finished, ApplicationData, ApplicationDataEmpty*

For the server part following inputs were used: *ClientHelloRSA, ClientHelloDHE, ClientKeyExchange, EmptyCertificate, ClientCertificate, ClientCertificateVerify, ChangeCipherSpec, Finished, ApplicationData, ApplicationDataEmpty*

For our research we only updated *cmd* section of configuration file and the output directory, so our resulting models are consistent with ones that were received by Ruiter et.al.

4 Applications

As mentioned in the previous sections the LearnLib algorithm was used to generate state models for the libreSSL, openSSL libraries and a Python TLS server, the specific versions are shown in table 1. LibreSSL was chosen as the initial candidate because it originated as a fork from the openSSL source code. The fork was created after the heartbleed bug[8] was discovered within openSSL. The aim of libreSSL is to modernize the codebase, improving security, and applying best practice development processes. The libreSSL application is still shipped with the openSSL client to improve compliance with current systems. Because of the difference in development of the two codebases, we deemed it interesting to verify the difference in state models.

In addition to the analysis of the libreSSL library, we also choose to generate a state model of a Python TLS server. As mentioned in the Python documentation, the TLS functionality of the Python programming language is actually managed by openSSL. A Python application actually passes the entire SSL functionality towards the openSSL library. Active state machine learning is an ideal algorithm to verify whether the usage of the openSSL library differs in any way when used in accordance with the Python programming language. The Python TLS server has been developed using the functions mentioned in the official Python documentation[9].

Library	**Version**	
libreSSL	2.2.3	http://www.libressl.org/
openSSL	1.0.1f	https://www.openssl.org/
Python TLS	2.7.9	https://docs.python.org/2/library/ssl.html

Table 1: List of the analyzed TLS libraries

5 Results

After generating the state models and analyzing the results, we discovered three anomalies within the state models. These anomalies, as far as we can determine, do not result in any security vulnerability within the associated TLS library. However, it might still be interesting to determine where these anomalies occur from.

[8] http://heartbleed.com/

[9] https://docs.python.org/2/library/ssl.html#server-side-operation

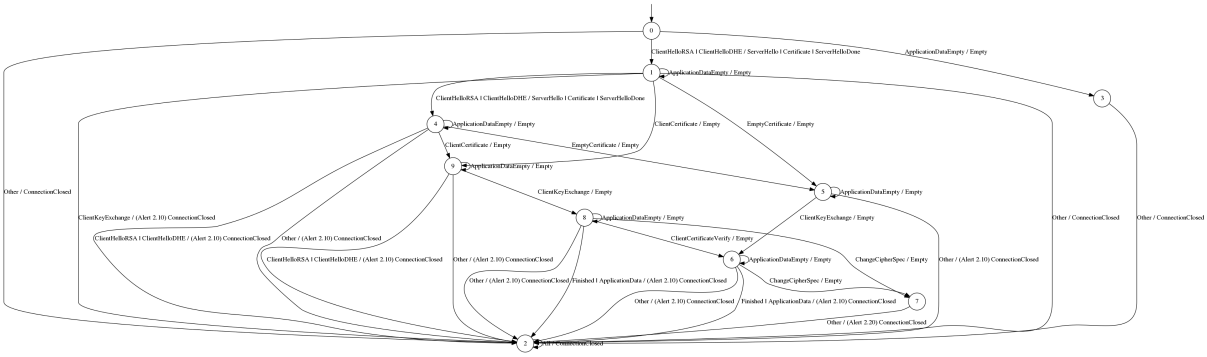


Figure 1: openSSL state model

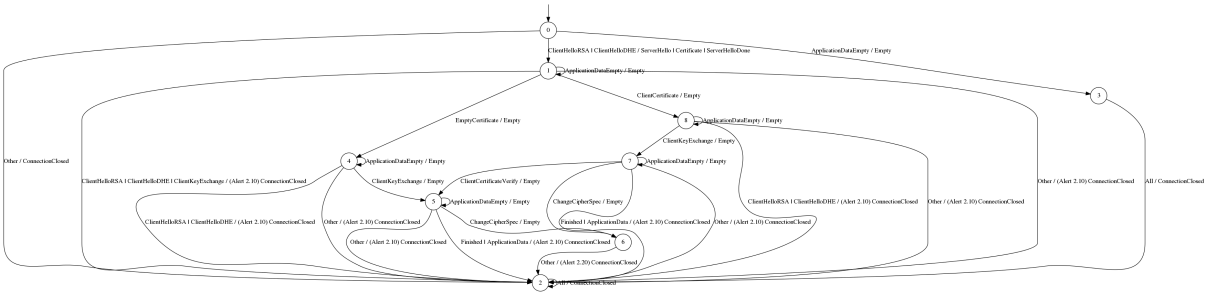


Figure 2: libreSSL state model

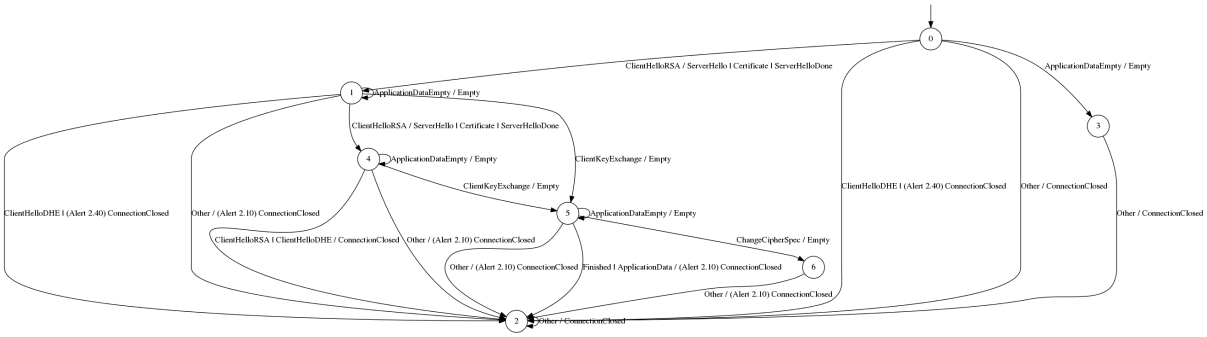


Figure 3: Python TLS server state model

5.1 Double ClientHello

One of the first discoveries was that the number of states are not equal between openSSL en libreSS state model. Further investigation discovered that this is because openSSL allows for a second *clientHello* message, this can be either a *clientHelloRSA* or a *clienthelloDHE* message. When comparing the two state models it is clear that openSSL allows for a second *clientHello* message which in turn will proceed to the normal states as the initial *clientHello* message. LibreSSL will result in a *ConnectionClosed* state when a second *clientHello* has been send. OpenSSL only allows for a second *clientHello* message and a third hello message will therefore, similarly as libreSSL, result in a *ConnectionClosed* state.

It is therefore possible that a potential attacker sends a second *ClientHello* message towards the server imposing as the user, however we deemed that this is not a security issues. Mainly because a potential attacker can already change all the attributes of the initial *ClientHello* message because this is send unencrypted over the network. Therefore there is not additional benefit for an attacker to send a second *ClientHello* message.

Although there are no security issues within this finding, it still can be discussed why this functionality is implemented within the openSSL library. It is not required, or stated, by the protocol to allow the second *ClientHello* message. We can therefore think of no reason why this functionality should be implemented and therefore advise to remove this functionality to improve code quality.

5.2 No ClientHelloDHE

As mentioned previously, the Python TLS functionality is using the openSSL library. It can therefore be expected that when state model of a Python TLS server is generated it will be equal to the state model of the openSSL library.

However as can be seen in figures 1 & 3, is that the state models are not equal. Actually there is a major difference between the two models. One of the first differences is that the Python TLS server does not allow the *ClientHelloDHE* message.

This is very strange behavior because this message should be available by default by the server. This message as can be seen in the state model of the openSSL library, can be used by default by the openSSL server. It is therefore strange behavior that the Python TLS server, which uses openSSL, does not enable this particular message. It should also be noted that a specific Alert status is returned, *Alert2.40*. However we could not find the specific message for this alert within the openSSL source code. Further investigation is therefore required to determine the exact nature of the alert.

5.3 No ClientKeyExchange

The another major difference between the state models of the Python TLS server and the openSSL library, is that the client authentication messages cannot be used with the Python TLS server. All messages are related to this action, *ClientCertificate, ClientCertificateVerify*, result in *ConnectionClosed* state.

Although this behavior is very different than the openSSL library, it can be discussed that this is due to an incorrect implementation of the Python application. The application was created using the official documentation, but no specific actions were taken to allow for client side authentication. This might be the cause why it is possible to use the client authentication messages. Further examination should therefore be performed in order to verify if this action is not allowed due to the implementation of the application or due to the Python TLS functionality.

6 Conclusion

Active state machine learning has again shown that it is a valuable research method in identifying security vulnerabilities and verifying the implementation of a protocol. Although the libraries which were analyzed are similar and implement the same protocol, there were some distinct differences. It has shown that there is a difference in implementing the SSL protocol, especially on how to react to double *ClientHello* messages, between the openSSL library and the libreSSL library.

In addition, it has shown that although the Python TLS functionality uses the openSSL library, there are some differences in how the protocol is actually being handled. Although these findings do not result in a direct security vulnerability, it is a clear distinction that there are major differences between the implementation of the same protocol.

These differences is import knowledge for a developer in order to verify their code quality and the correct implementation of the protocol. It is therefore highly recommended to continue the research on the state models of different TLS applications and implementations.

References

[1] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[2] O. Bauer, M. Geske, and M. Isberner. Analyzing program behavior through active automata learning. *International Journal on Software Tools for Technology Transfer*, 16(5):531–542, 2014.

[3] C.Y. Cho, D. Babi, C.E.R. Shin, and D. Song. Inference and analysis of formal models of botnet command and control protocols. pages 426–439, 2010.

[4] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.*, 4(3):178–187, May 1978.

[5] J. de Ruiter and E. Poll. Protocol state fuzzing of tls implementations. 2015.

[6] Y. Hsu, G. Shu, and D. Lee. Model-based approach to security flaw detection of network protocol implementations. pages 114–123, 2008.

[7] S. Khattak, N.R. Ramay, K.R. Khan, A.A. Syed, and S.A. Khayam. A taxonomy of botnet behavior, detection, and defense. *IEEE Communications Surveys and Tutorials*, 16(2):898–924, 2014.

[8] L. Zhiqiang, J. Xuxian, X. Dongyan, and Xiangyu Z. Automatic protocol format reverse engineering through context-aware monitored execution. 2008.