Testing TLS Using Combinatorial Methods and Execution Framework

Dimitris Simos, Josip Bozic, Feng Duan, Bernhard Garn, Kristoffer Kleine, Yu Lei, Franz Wotawa

Testing TLS Using Combinatorial Methods and Execution Framework

Dimitris E. Simos[1], Josip Bozic[2], Feng Duan[3], Bernhard Garn[1], Kristoffer Kleine[1], Yu Lei[3], and Franz Wotawa[2]

[1] SBA Research, A-1040 Vienna, Austria
{bgarn,kkleine,dsimos}@sba-research.org
[2] Institute for Software Technology, Graz University of Technology, A-8010 Graz, Austria
{jbozic,wotawa}@ist.tugraz.at
[3] University of Texas at Arlington, Arlington, TX 76019, USA
feng.duan@mavs.uta.edu, ylei@cse.uta.edu

Abstract. The TLS protocol is the standard for secure Internet communication between two parties. Unfortunately, there have been recently successful attacks like DROWN or BREACH that indicate the necessity for thoroughly testing TLS implementations. In our research work, we focus on automated test case generation and execution for the TLS security protocol, where the aim is to make use of combinatorial methods for providing test cases that ideally also reveal previously unknown attacks. This is made feasible by creating appropriate input parameter models for different messages that can appear in a TLS message sequence. In this paper, we present the resulting test case generation and execution framework together with the corresponding testing oracle. Furthermore, we discuss first empirical results obtained using different TLS implementations and their releases.

Keywords: combinatorial testing, security testing, security protocols, TLS.

1 Introduction

Software implementations of the Transport Layer Security (TLS) protocol specification are a critical component for the security of the Internet communications and beyond. Software bugs and attacks, such as Heartbleed, DROWN, BREACH and POODLE, still surface in implementations of the TLS protocol, despite many years of analysis (at least for open-source implementations). This can be attributed to the complexity of the protocol and its very high number of interactions.

In this paper, we describe a security testing technique for the TLS protocol. Since the TLS handshake protocol is one of the most important components of TLS, our approach focuses on this part of the protocol in order to test TLS implementations for potential security leaks. After providing some basic definitions, we describe the input model for combinatorial test case generation. Note

that this is the first time where combinatorial testing is applied to the testing of the TLS protocol.

For this approach, three different input models are constructed, each for one client-side TLS event, respectively. Every of these events encompasses a specific set of possible parameters and values. In addition to the modelling part, the practical part deals with the implementation of a test execution framework. This has the possibility to communicate, and thus, to attack TLS implementations in an automated manner. The framework comprehends all TLS events according to the TLS standard and can execute a handshake for this purpose.

In particular, the paper is structured as follows: Section 2 describes related work while Section 3 defines preliminary concepts for the methodology developed in this paper. In Section 4 we depict a combinatorial approach for testing TLS which was made feasible by developing the necessary input models and give examples of the generated test sets. Moreover, Section 5 describes the test execution framework we have developed for TLS together with a test oracle. Finally, Section 6 presents the results of the executions followed by a detailed evaluation and Section 7 concludes the paper.

2 Related Work

New activities in security testing were sparked by [11], in which the authors found several critical vulnerabilities in commonly used TLS implementations. The identified bugs were classified as state machine bugs and this put the internal state machines of TLS implementations into the focus, next to and on the same level of criticality as the implementation of pure cryptographic functionality. The systematic testing of TLS implementations based on the principles of these kind of state machine bugs with a tool was presented in [32] and offers the user the possibility to create custom TLS message flows and arbitrarily modify message contents. The strategy for testing of TLS implementations via the exchange of modified messages or injection attacks was also followed in [28] and [12].

The works [30][11] follow a model-based approach for testing TLS implementations, using [29] to extract the respective state machines.

The certificate validation logic in SSL/TLS implementations was tested in [16], and a cross-protocol attack was presented in [26]. Attacks on authentication were presented in [13] and [14]. A plaintext recovery attack was given in [9]. The usage of invalid curves can lead to practical attacks as shown in [21], and new Bleichenbacher side channels and attacks were presented in [27]. A systematic analysis of the security of TLS can be found in [22].

Testing of the newly proposed TLS 1.3 [7] was discussed in [18]. Finally, care has to be taken to compare in detail the security properties that TLS provides and the requirements of any application using it [10].

3 Preliminaries

In this section we detail some preliminary concepts needed for the work carried out in this paper. In particular, we link expressions from the domain of software testing to expressions from the domain of protocol specification. Regarding terminology for protocol specification, we use and refer to RFC 5246, "The Transport Layer Security (TLS) Protocol, Version 1.2" [4].

In their union collectively simply referred to as the *TLS protocol 1.2*, its specification [4] actually defines multiple protocols, some of which operate on top of each other. In this paper, we focus on the *Handshake Protocol*, a member of the *TLS Handshaking Protocols*. The semantics encapsulated in the *Handshake Protocol* are coded in the exchange of *messages* between a client and a server. If a client initially connects to a server and wants to establish a secure connection, it will follow the *Handshake Protocol* and start by sending a `ClientHello` message. According to the specification, the protocol semantics translate into an ordered sequence of Handshake messages that allow a client and server to establish a *secure connection* and start exchanging application data.

The software testing view on the *Handshake protocol* is that it regards these messages as *abstract* events, which appear as ordered sequences in practice. Handshake messages appear in practice with all of their values instantiated during the execution of the Handshake protocol. We only consider protocol compliant sequences of abstract messages. To be able to execute derived test cases from these abstract events, it is necessary to define an *input model* for these abstract events corresponding to Handshake messages. We aim to develop such input models using combinatorial methods for each of these abstract events (see Section 4.2), which will be used to instantiate concrete Handshake messages.

We model each abstract event independently and also want to test them individually,but nevertheless we are interested in always continuing the Handshake protocol as long as possible, trying to do a complete Handshake. To that end, we define a standard instantiated Handshake message for each considered client-side Handshake message. These intentionally "harmless" Handshake messages will be used create as complete as possible concrete sequences of Handshake messages. In other words, during our testing it is necessary to distinguish between Handshake messages populated with values chosen for testing one specific Handshake message (whose purpose is to disrupt the normal execution work flow via their values) and template Handshake messages (whose single purpose is to move the current status of Handshake during protocol execution to a particular state). Please note that in any sequence there appears exactly one message, which is derived from the abstract model, while all other messages are template messages (see also Section 6). To sum up, we are only actively sending non-standard client messages, and as a result we are testing the server side of the two communicating parties.

4 Combinatorial Testing

Combinatorial Testing (CT) is a widely applicable methodology and technology for software testing. In a combinatorial test plan, all interactions between parameters up to a certain level are covered. For example, in pairwise testing, for every pair of parameters, every pair of values will appear at least once. Furthermore, a CT strategy, called t-way testing, requires every combination of values of any t parameters to be covered by at least one test, where t is referred to as the strength of coverage and usually takes a small value (1 to 6) [24]. Each combination of values of a set of parameters is considered to represent one possible interaction among these parameters.

4.1 Application to Testing of TLS

Even though combinatorial testing is a proven methodology for security testing [31], this is the first time where such a combinatorial approach is being used to test the TLS protocol.

When applying CT to testing TLS, we focus on the possible interactions among parameters of TLS messages. An interaction may accur among those parameters in the same TLS message, or among parameters in different TLS messages. Given a sequence of TLS messages, to capture these interactions among its parameters, we first considered a naive approach: flat CT. It means all parameters of these messages are listed in one model flatly, and every combination of values of any t parameters will be covered. This approach is simple, however, when TLS event sequence is changed, it requires to modify the whole model of flat CT and redo test case generation.

In order to make CT more flexible for multiple TLS event sequences, here we choose another approach: hierarchical CT. That is to say, there are two levels for CT test case generation: intra-message level (lower level) and inter-message level (upper level). On intra-message level, CT will generate t-way test cases for parameters of a single message (client-side). For each type of message, an intra-message t-way test set will be generated from its model separately. On inter-message level, each type of message is expressed as a parameter, and its intra-message t-way test cases are expressed as parameter values. Given a sequence of TLS messages, its messages as parameters and their intra-message t-way test cases as parameter values will then be used to generate inter-message t'-way test cases. Note that, t' can be different from t, e.g., by now, we use 1-way on inter-message level which means CT only be applied on single message.

4.2 Input Parameter Modeling

Since CT creates test cases by selecting values for input parameters and by combining these values, Input Parameter Modeling (IPM) is required to capture the input test space for real-life applicability of CT. IPM mainly works on identifying the parameters, values, and the relations of parameters. According to [17] [20], a list of TLS messages can be cataloged, and IPM based on their

data structures can be made. In TLS protocol, each message is constructed from two layers: TLS Record Layer (bottom layer), and Protocol Message Layer (top layer, as shown in Table 1, note that Message Body only exists for Handshake Protocol Messages). Since other attributes in these two layers will be determined by values in Message Body, we only need to do IPM on attributes in Message Body for handshake protocol messages.

Table 1. Protocol Message Layer (Handshake Protocol)

Handshake Protocol Message		
Message Type (1byte, 0x00-0xFF)	Length of message body (3bytes, uint24)	Message Body
hello_request(0)	0	empty
client_hello(1)	the length of message body in bytes	ClientHello
server_hello(2)	the length of message body in bytes	ServerHello
certificate(11)	the length of message body in bytes	Certificate
server_key_exchange(12)	the length of message body in bytes	ServerKeyExchange
certificate_request(13)	the length of message body in bytes	CertificateRequest
server_hello_done(14)	0	empty
certificate_verify(15)	the length of message body in bytes	CertificateVerify
client_key_exchange(16)	the length of message body in bytes	ClientKeyExchange
finished(20)	the length of message body in bytes	Finished
(255)		

In this paper, since we only test TLS implementation from client side, IPM is only applied on client-side handshake messages. For example, we do IPM on `ClientHello` message, but not on `ServerHello` message. Here, we list general TLS messages as Table 2, which are used for handshake procedure in Figure 1. And among all nine messages, only M1, M5, M6 and M7 are client-side messages. Note that M6 is CCS protocol message, not Handshake. According to that, we only have to do IPM on the parameters of M1, M5 and M7.

Table 2. General TLS messages

M1	ClientHello
M2	ServerHello
M3	ServerCertificate
M4	ServerHelloDone
M5	ClientKeyExchange
M6	ClientChangeCipherSpec
M7	ClientFinished
M8	ServerChangeCipherSpec
M9	ServerFinished

Based on the TLS protocol specification, we can derive parameters and possible values for M1, M5 and M7. In practical terms, parameter values may be abstracted and limited in domain size, while some parameters should be subdivided into meta-parameters. For example, in M1, a ClientUnixTime meta-parameter is

extracted from "client random" parameter, and assigned $RealTime \pm x$ as part of its abstract values, for detecting potential vulnerability on time processing mechanism of TLS implementations.

A challange is that TLS involves a lot of cipher/compression/Hash/PRF functions. When a handshake message includes a collection (list) of these options, some parameters cannot enumerate all their possible values but only give some representative values. For example, cipher suites of `ClientHello` message can be any list of the cryptographic options supported by the client. In order to avoid unnecessarily huge domain size for the *cipher suites* parameter, here we only use single cipher suites as its values, but not non-singular lists of suites. Note that, this shortcut may miss some potential interactions since parameter values are limited.

As mentioned earlier, for any TLS standard sequence containing client-side messages: M1, M5 and M7, we first conduct an IPM on these messages, and then create input model for a system under test into the ACTS tool [33]. ACTS is a widely-used tool for generating CT test cases which supports multi-way coverage [25], constraints [34] [23], and is well optimized [19].

The input model of M1 can be listed as follows:

```
Protocol Version: TLS10,TLS11,TLS12,DTLS10,DTLS12
Client Unix Time: RealTime,RealTime-x,RealTime+x
Client Random: 28-byteRand
Session ID: NULL,32-byteID
Supported Cipher Suites: TLS_FALLBACK_SCSV,
   TLS_NULL_WITH_NULL_NULL,TLS_RSA_WITH_NULL_SHA256,
   TLS_RSA_WITH_AES_128_CBC_SHA256,
   TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA
Supported Compression Methods: NULL,DEFLATE,LZS
```

Similarly, the input model of M5 can be listed as follows:

```
KeyExchangeAlgorithm: rsa,dhe_dss,dhe_rsa,dh_dss,dh_rsa
   ,dh_anon
ClientProtocolVersion: TLS10,TLS11,TLS12,DTLS10,DTLS12
ClientRandom: 46-byteRand
PublicValueEncoding: implicit,explicit
Yc: empty,ClientDiffie-HellmanPublicValue
```

Note that, if KeyExchangeAlgorithm is rsa, an EncryptedPreMasterSecret key will be concreted by ClientProtocolVersion and ClientRandom; otherwise, a ClientDiffieHellmanPublic value will be concreted when PublicValueEncoding is explicit.

Also, the input model of M7 can be listed as follows:

```
master_secret: empty,half,default,changebyte,multiply
finished_label: client finished
Hash: empty,half,default,changebyte,multiply
```

Both master_secret and Hash meta-parameters have the same values. These abstract values represent operations to be performed on already dynamically calculated values (from the previous message flow), which are required for concreting a real value for "verify_data" parameter. More details are mentioned in next section.

4.3 Pairwise Test Suites for TLS Testing

With input models for M1, M5 and M7, ACTS can easily generate intra-message test cases for each message, e.g., pairwise (2-way) test cases of M1 are shown in Table 3. For brevity, we only show 10 tests in table, while there are total 25 pairwise tests for input model of M1. By using ACTS, we also generate 30 pairwise tests for M5 and 25 pairwise tests for M7. Note that, pairwise test set may not be able to trigger all possible failures during message processing, but it is a good starting point for applying CT on TLS.

Table 3. Pairwise Test Cases of M1

Protocol Version	Client Unix Time	Client Random	Session ID	Supported Cipher Suites	Supported Compression Methods
TLS10	RealTime-x	28-byteRand	32-byteID	TLS_FALLBACK_SCSV	DEFLATE
TLS10	RealTime+x	28-byteRand	NULL	TLS_NULL_WITH_NULL_NULL	LZS
TLS10	RealTime	28-byteRand	32-byteID	TLS_RSA_WITH_NULL_SHA256	NULL
TLS10	RealTime-x	28-byteRand	NULL	TLS_RSA_WITH_AES_128_CBC_SHA256	LZS
TLS10	RealTime+x	28-byteRand	32-byteID	TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA	NULL
TLS11	RealTime	28-byteRand	NULL	TLS_FALLBACK_SCSV	LZS
TLS11	RealTime-x	28-byteRand	32-byteID	TLS_NULL_WITH_NULL_NULL	NULL
TLS11	RealTime+x	28-byteRand	NULL	TLS_RSA_WITH_NULL_SHA256	DEFLATE
TLS11	RealTime	28-byteRand	32-byteID	TLS_RSA_WITH_AES_128_CBC_SHA256	DEFLATE
TLS11	RealTime-x	28-byteRand	NULL	TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA	LZS
...	...	...	...	...	...

5 Execution Framework

In this section we give an overview about the execution framework for security testing of TLS implementations. In order to execute the test cases, we developed a framework that reads the generated test sets and executes them against any implementation of TLS. The result is an execution method that offers the ability to configure and execute test cases in an automated manner.

The framework itself builds upon TLS-Attacker [6], an implementation for analyzing TLS libraries. The initial implementation encompasses the possibility to establish a TLS handshake and to specify individual parameters for this purpose. However, in our case we adapted the framework in order to test TLS implementations according to concrete values coming from combinatorial testing.

Figure 1 depicts one handshake procedure of the TLS protocol. It encompasses a set of client and server side messages that are exchanged between the peers. Each messages is called a TLS event. During the translation process from abstract to concrete test cases, several values are generated dynamically, whereas other values is prespecified. If no "unexpected" behavior is encountered during the handshake, a hopefully secure connection is established.

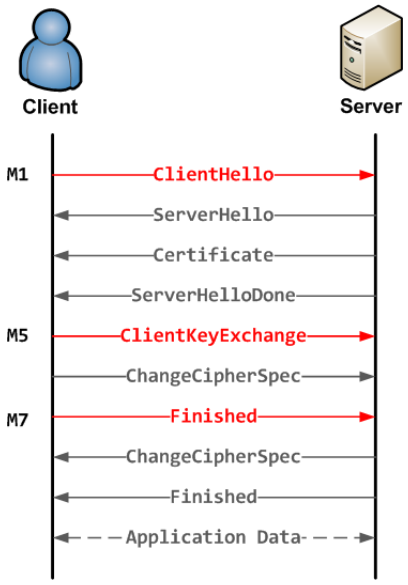

Fig. 1. TLS handshake and tested messages.

However, since the intention in our approach lies on detecting abnormal behavior of TLS implementations, two different approaches can be taken. First, the order of TLS events can be manipulated, thus deviating from the default sequence. This approach was demonstrated in a simple example in [15]. Another option would be to check the default sequence by manipulating the concrete parameter values of some of the individual TLS events. In this way, we might provoke a reaction from the server, which could lead to different behavior of the tested SUTs.

In this paper, parameter values of client-side TLS events are changed dynamically. An abstract overview of the functionality of the implementation is given in Figure 2. First, the generated concrete values are read by the test execution framework (TEF). Actually, we execute the TLS handshake as one test case. In addition to the message flow, one specific type of client-side message is tested per test case. For example, first we test whether manipulating the `ClientHello` message, but keeping the default dynamically generated values of the other messages will result on a termination of the handshake procedure. Then, values for `ClientKeyExchange` are manipulated while all other TLS events are processed via standard messages (see Section 3). Finally, client's `Finished` message reads the generated values from the combinatorial test sets.

5.1 Test Oracle

Constructing the test oracle for security testing of protocols has been realized in the following way. Specifying an expected output value on the concrete level

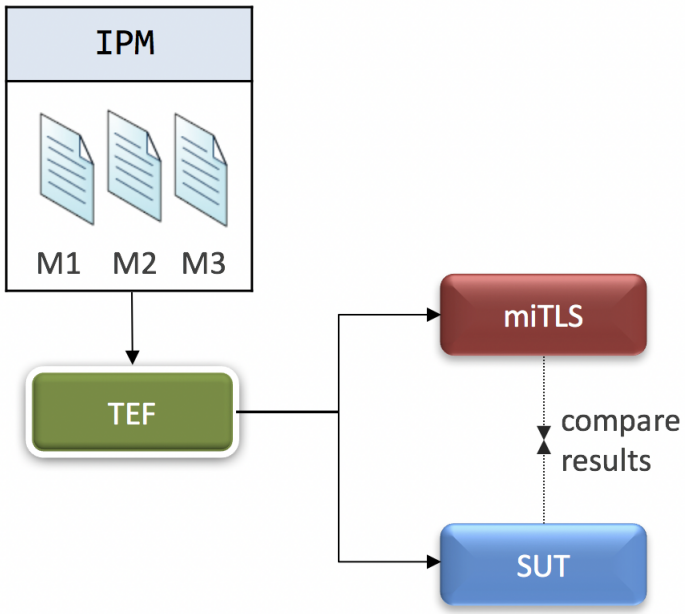

Fig. 2. The test execution framework and its environment.

is hard to define because of the complexity of the TLS protocol in general. Moreover, every TLS implementation has its own mechanisms and could react in a different way to the same input values. Because of this, our approach defines the oracle on an abstract level. Since the message flow of the TLS handshake is known, we define the oracle based on the flow of TLS events of a reference implementation. For this cause we have selected as reference implementation the **miTLS**, where some security properties have been proven correct. Every test set is executed against miTLS, where all sent and received messages are recorded. Afterwards, the same test set is used on the individual SUTs, thereby recording their corresponding replies. In the aftermath, as depicted in Figure 2, we compare execution traces from both the reference implementation and the individual SUT.

5.2 Testing Procedure

In order to give a demonstration, let's discuss a test case where we test `ClientHello`. As mentioned, the concrete values can be read in their current form or they represent an operation that needs to be processed on a parameter. The first test case from the pairwise test set for M1 (see Table 3) reads as follows:

```
TLS10,RealTime-x,28-byteRand,32-byteID,TLS_FALLBACK_SCSV,DEFLATE
```

The values indicate that the client offers TLS 1.0 as its supported protocol, whereas `TLS_FALLBACK_SCSV` and `DEFLATE` are picked as the cipher suite and

compression method, respectively. On the other hand, the value for Unix time indicates an operation on a value. Here the current real-time is subtracted by the value `x`, i.e. an amount of 10, which stands for years. Additionally, the values `28-byteRand` and `32-byteID` instruct not to make any changes but to use the agreed values for the random number and session ID. Finally, the framework generates the following message according to the values:

```
Protocol Version: TLS10
Client Unix Time: Sat Jun 09 04:33:32 CEST 2007
Client Random: 01 4B ...
Session ID:
Supported Cipher Suites: 56 00
Supported Compression Methods: 01
```

In our execution framework, each of the parameters is dealt with individually. This means that the program can distinguish between assigned values and operations. As depicted above, the cipher suite and compression methods are defined in their unique hexadecimal form of their byte representations.

The functionality of `ClientKeyExchange` is different from M1 in the sense that more of their values are generated in run-time. Take an example from the first row in M5:

```
rsa,TLS10,46-byteRand,explicit,ClientDiffie-HellmanPublicValue
```

This IPM differs from the others with regard to the fact that it does not encompass one fixed type of message. According to the chosen key exchange algorithm either a RSA-encrypted premaster secret message or Diffie-Hellman public value. Both message types have specific parameters. Whereas the first covers the client protocol version and random value for the pre-master secret, the second encompasses the public value `Yc`. In case that the client has already sent the public value in the certificate, the public value encoding is **implicit**. Otherwise, **explicit** instructs to send `Yc` inside the new message. A RSA-encrypted message with the previous value for `KeyExchangeAlgorithm` might have the following looks:

```
KeyExchangeAlgorithm: rsa
ClientProtocolVersion: TLS10
ClientRandom: C5 FE ...
```

In contrast to both examples above, `Finished` is fully generated according to run-time values. By reading the values from the M7's IPM we get:

```
half,client finished,changebyte
```

Usually the master secret represents a 48-byte value that is calculated during the handshake. The Hash is calculated on-the-fly as well by taking into consideration previous TLS events, thus producing a 32-byte value. The finished label will always remain the same. In order to give an example, let's assume that the obtained **master_secret** looks the following way:

```
59 7A E7 37 A6 C9 18 90  C9 C7 99 44 57 FE 06 BC
CF 20 A3 DE 12 56 3B DE  12 AE 10 B4 2E CB 06 61
8C DC 96 FE 77 07 37 B7  E9 73 D5 93 32 E6 9E 9D
```

The 32-byte value of **Hash** looks like:

```
C5 11 5E C7 56 7A 9A E2  2A 1F 9B F3 38 5D FB 08
38 D0 31 B5 D3 B7 35 42  13 F2 64 58 12 26 92 A9
```

The concrete values indicate that the first byte array has to be cut in half, i.e. using only half of its value further. On the other hand, `changebyte` will perform an operation where the first byte of the calculated hash will be exchanged by the byte `0xFF`. Finally, the resulting message will have the following values:

```
master_secret:
  59 7A E7 37 A6 C9 18 90  C9 C7 99 44 57 FE 06 BC
  CF 20 A3 DE 12 56 3B DE
finished_label: client finished
Hash:
  FF 11 5E C7 56 7A 9A E2  2A 1F 9B F3 38 5D FB 08
  38 D0 31 B5 D3 B7 35 42  13 F2 64 58 12 26 92 A9
```

By changing these two values we intend to surprise the system by submitting similar but malformed expected values. In such way, the SUT might be tricked into unexpected behaviour.

6 Evaluation

Since the execution framework is meant for testing of TLS implementations, five different programs have been installed for this purpose. As already mentioned, the reference implementation is miTLS ver. 0.9 [2], whereas the other tested SUTs are OpenSSL ver. 1.0.1e [3], wolfSSL ver. 3.10.2 [8], mbed TLS ver 2.4.2 [5] and GnuTLS ver 3.5.9 [1]. As noted before, three client-side messages have been tested with 25, 30 and 25 inputs, respectively. The obtained results are depicted in Table 4. The results show the results from three different SUT, which are categorized according to three types of outputs. The first column depicts how many times a complete handshake (comp) could be established between the framework and the SUT. Additionally, we can see how many times the attempt was fully rejected (reject), i.e. by getting no or only one response (for example an alert message) from the server. Finally, the number of incomplete handshakes (incomp) depicts the situation where no handshake was established but the attempt was not initially rejected either.

The goal of the framework is to offer an automated linkage between combinatorial testing and protocol testing, i.e. it should be able to test any TLS implementation by requiring only minor tester interaction. Usually only the port is changed in order to access another SUT. As explained before, the default TLS

Table 4. Evaluation Results

SUT	miTLS			OpenSSL			mbed TLS		
	comp	**reject**	**incomp**	**comp**	**reject**	**incomp**	**comp**	**reject**	**incomp**
M1	0	25	0	1	24	0	1	17	7
M5	0	30	0	0	0	30	0	0	30
M7	0	25	0	1	0	24	1	0	24

handshake is executed by manipulating values of the emphasized client-side messages. No additional preferences are set. We want to examine whether for the same inputs, different output traces will be achieved.

The testing proceeds in the following way. As mentioned earlier, there are three test scenarios and in each of them we want to execute the entire TLS handshake. When testing M1, the values from its IPM will be used, whereas M5 and M7 will have standard on-the-fly generated values. Also, when testing M5, we use standard messages for M1 and M7. Finally, for M7 the framework does not make use of the IPM for M1 and M5 since it uses similarly standard messages for M1 and M5. This means that we are testing the whole handshake in each scenario but change values only for one TLS event. In such way, we want to see how the manipulation of one single TLS event affects the overall handshake.

When testing M1, OpenSSL does not continue with the handshake procedure for any of the submitted values by throwing a decode error. This may happen because of protocol version or cipher suite negotiation. However, mbed TLS does return more results, even succeeding in establishing a handshake between both peers. In other cases, a `BAD_RECORD_MAC` is thrown after the client's `Finished` message. This leads to the conclusion that because of specific concrete values for M1 a different behavior from the recorded one for miTLS is triggered after a few messages. Investigating these values might be of interest for the tester. In other cases, the execution terminates after the `ServerHelloDone` message. The rest of the results usually indicates cipher suite negotiation failures and terminates the execution. This is as well the case with wolfSSL, where the handshake usually breaks up with a `SSL_accept` failed error. This is the same case with miTLS and GnuTLS, which fail because of diverse `HANDSHAKE_FAILURE`s or message MAC verification failures. In general, it seems that miTLS resists handshake attempts by far more than all other applications, as we *expected* it should.

For M5, OpenSSL as well as mbed TLS an error is thrown after server's hello done message. Also, wolfSSL cannot match any cipher suite, whereas GnuTLS throws `HANDSHAKE_FAILURE`s. The reasons for the results from the first tested SUTs are problems with the specific key exchange messages. In some cases, errors are triggered because the SUT expects a RSA-encrypted message. This usually leads to the termination of the handshake procedure. In summary, testing for M5 seems to produce most difficulties for all SUTs.

Finally, the test results for M7 are as follows. OpenSSL usually reaches a state where client's `Finished` message is received, after which an alert terminates the execution. The operations done on the dynamically generated values for

master_secret and **Hash** are detected by the SUT, which causes the breakup. However, in one case a handshake was finalized successfully. Similar results are obtained for mbed TLS. On the contrary, miTLS and wolfSSL do not conclude any handshake, which is the case with GnuTLS as well.

In general, some of the obtained results are similar. For example, two different SUTs behave similar when rejecting the same input. Whereas miTLS rejects any further communication with an exception, OpenSSL sends an additional alert message. Although the behavior is not quite the same, it can be concluded that both applications react slightly different when making the same decision.

7 Conclusion

In this paper we generated combinatorial test sets which include dynamically assigned values of individual TLS events. During execution, several abstract messages are generated on-the-fly and populated with concrete data using combinatorial (testing) strategies. During the testing process, the framework tests different SUTs by focusing separately on three of the TLS handshake messages. Every of these events is tested as part of the standard handshake procedure but with manipulated values according to the combinatorial input model. Then, we compared the resulting execution traces to the submitted input and to the results of other SUTs.

The framework was able to test TLS implementations in an automated manner. Also, different test results have been achieved with regards to concrete input values. The analysis of these results and the causes for misbehavior represents an issue for the tester in order to track whether a vulnerability could be detected. In general, we were able to identify test cases that led to non-uniform behavior of TLS implementations. Generating more of such test cases would represent a promising task for the future.

We draw the conclusion that the developed framework and oracle is strong enough to distinguish different behavior among TLS implementations. The obtained results require more investigation in order to track the cause of this behavior and examine whether security leaks have occurred.

However, in order to reach more fine-grained testing results, more effort has to be put in the strengthening the test oracle and generating test cases according to a higher combinatorial strength. In this case, more diverse test cases would be generated that could lead to more thorough testing and make another step towards security testing of TLS. As future work, we plan the framework to be extended further to ease the usability for a tester and provide feedback on the internal processing during execution.

Acknowledgement

The research presented in the paper has been funded in part by the Austrian Research Promotion Agency (FFG) under grant 851205 (Security ProtocoL Interaction Testing in Practice - SPLIT) and the Austrian COMET Program (FFG).

References

1. The gnutls transport layer security library. http://www.gnutls.org/, accessed: 2017-06-07
2. mitls: A verified reference implementation of tls. https://mitls.org/, accessed: 2017-06-07
3. Openssl. https://www.openssl.org/, accessed: 2017-06-07
4. Rfc 5246. https://tools.ietf.org/rfc/rfc5246.txt, accessed: 2017-05-31
5. Ssl library mbed tls / polarssl. https://tls.mbed.org/, accessed: 2017-06-07
6. Tls-attacker. https://github.com/RUB-NDS/TLS-Attacker, accessed: 2016-12-04
7. The transport layer security (tls) protocol version 1.3. https://tools.ietf.org/html/draft-ietf-tls-tls13-07, accessed: 2017-05-31
8. wolfssl. https://www.wolfssl.com/, accessed: 2017-06-07
9. AlFardan, N., Paterson, K.G.: Plaintext-recovery attacks against datagram tls. In: Network and Distributed System Security Symposium (NDSS 2012) (2012)
10. Berbecaru, D., Lioy, A.: On the robustness of applications based on the ssl and tls security protocols. In: European Public Key Infrastructure Workshop. pp. 248–264. Springer (2007)
11. 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: Proceedings of the 36th IEEE Symposium on Security and Privacy (2015)
12. Beurdouche, B., Delignat-Lavaud, A., Kobeissi, N., Pironti, A., Bhargavan, K.: Flextls: A tool for testing tls implementations. In: 9th USENIX Workshop on Offensive Technologies (WOOT'15) (2015)
13. Bhargavan, K., Lavaud, A.D., Fournet, C., Pironti, A., Strub, P.Y.: Triple handshakes and cookie cutters: Breaking and fixing authentication over tls. In: Security and Privacy (SP), 2014 IEEE Symposium on. pp. 98–113. IEEE (2014)
14. Bhargavan, K., Leurent, G.: Transcript collision attacks: Breaking authentication in tls, ike, and ssh. In: Network and Distributed System Security Symposium–NDSS 2016 (2016)
15. Bozic, J., Kleine, K., Simos, D.E., Wotawa, F.: Planning-based security testing of the ssl/tls protocol. In: Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) (2017)
16. Brubaker, C., Jana, S., Ray, B., Khurshid, S., Shmatikov, V.: Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations. In: Proceedings of the 2014 IEEE Symposium on Security and Privacy (2014)
17. Dierks, T., Rescorla, E.: Rfc 5246: The transport layer security (tls) protocol. The Internet Engineering Task Force (2008)
18. Dowling, B., Fischlin, M., Günther, F., Stebila, D.: A cryptographic analysis of the tls 1.3 handshake protocol candidates. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 1197–1210. ACM (2015)
19. Duan, F., Lei, Y., Yu, L., Kacker, R.N., Kuhn, D.R.: Optimizing ipog's vertical growth with constraints based on hypergraph coloring. In: Software Testing, Verification and Validation Workshops (ICSTW), 2017 IEEE International Conference on. pp. 181–188. IEEE (2017)
20. Hollenbeck, S.: Transport layer security protocol compression methods (2004)
21. Jager, T., Schwenk, J., Somorovsky, J.: Practical invalid curve attacks on tls-ecdh. In: European Symposium on Research in Computer Security. pp. 407–425. Springer (2015)

22. Krawczyk, H., Paterson, K.G., Wee, H.: On the security of the tls protocol: A systematic analysis. In: CRYPTO (2013)
23. Kuhn, D.R., Bryce, R., Duan, F., Ghandehari, L.S., Lei, Y., Kacker, R.N.: Chapter one-combinatorial testing: Theory and practice. Advances in Computers 99, 1–66 (2015)
24. Kuhn, R., Lei, Y., Kacker, R.: Practical combinatorial testing: Beyond pairwise. It Professional 10(3) (2008)
25. Lei, Y., Kacker, R., Kuhn, D.R., Okun, V., Lawrence, J.: Ipog/ipog-d: efficient test generation for multi-way combinatorial testing. Software Testing, Verification and Reliability 18(3), 125–148 (2008)
26. Mavrogiannopoulos, N., Vercauteren, F., Velichkov, V., Preneel, B.: A cross-protocol attack on the tls protocol. In: ACM CCS 12: 19th Conference on Computer and Communications Security (2012)
27. Meyer, C., Somorovsky, J., Weiss, E., Schwenk, J., Schinzel, S., Tews, E.: Revisiting ssl/tls implementations: New bleichenbacher side channels and attacks. In: USENIX Security. vol. 14, pp. 733–748 (2014)
28. Morais, A., Martins, E., Cavalli, A., Jimenez, W.: Security protocol testing using attack trees. In: CSE (2), IEEE Computer Society (2009). pp. 690–697 (2009)
29. Raffelt, H., Steffen, B., Berg, T.: Learnlib: A library for automata learning and experimentation. In: Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'05). pp. 62–71 (2005)
30. de Ruiter, J., Poll, E.: Protocol state fuzzing of tls implementations. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 193–206 (2015)
31. Simos, D.E., Kuhn, R., Voyiatzis, A.G., Kacker, R.: Combinatorial methods in security testing. IEEE Computer 49, 40–43 (2016)
32. Somorovsky, J.: Systematic fuzzing and testing of tls libraries. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16) (2016)
33. Yu, L., Lei, Y., Kacker, R.N., Kuhn, D.R.: Acts: A combinatorial test generation tool. In: Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on. pp. 370–375. IEEE (2013)
34. Yu, L., Lei, Y., Nourozborazjany, M., Kacker, R.N., Kuhn, D.R.: An efficient algorithm for constraint handling in combinatorial test generation. In: Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on. pp. 242–251. IEEE (2013)