





Applying Symbolic Execution to Test Implementations of a Network Protocol Against its Specification

Hooman Asadian , Paul Fiterău-Broștean , Bengt Jonsson , Konstantinos Sagonas 
 Department of Information Technology, Uppsala University, Uppsala, Sweden
 {hooman.asadian, paul.fiterau_brostean, bengt, kostis}@it.uu.se

Abstract—Implementations of network protocols must conform to their specifications in order to avoid security vulnerabilities and interoperability issues. We describe our experiences using symbolic execution to thoroughly test several implementations of a network security protocol against its specification. We employ a methodology in which we first extract requirements from the protocol’s RFC and turn them into formulas. These formulas are then utilized by symbolically executing the protocol implementation to explore code paths that can be traversed on packet sequences that violate a requirement. When this exploration exposes a bug, corresponding input values are produced and turned into test cases that can validate the bug in the original implementation. Since we let symbolic execution be guided by requirements, it can naturally produce a wide variety of requirement-violating input sequences, which is difficult to achieve with existing techniques for protocol testing. We applied this methodology to test four different implementations of DTLS against the protocol’s RFC. We were able to quickly expose a known CVE in an older version of OpenSSL, and to discover numerous previously unknown vulnerabilities and non-conformance issues in DTLS implementations, which have by now been confirmed and fixed by their implementors.

Index Terms—symbolic execution, network security testing

I. INTRODUCTION

Implementations of network protocols, such as TCP, TLS, DTLS, etc., must conform to their specifications in order to avoid security vulnerabilities and interoperability issues. Even seemingly innocent deviations from the standard specification may open implementations up for security attacks. Examples include Heartbleed [8] and the TLS POODLE [3], [23] downgrade vulnerability enabled by insufficient checking of length fields or version numbers in input packets. At the same time, testing of protocol implementations is made difficult by the fact that they are *stateful*. To test the processing of a particular packet, the implementation must first be brought to a specific state by an appropriate packet sequence. Moreover, the requirements concerning that packet may depend on the preceding sequence. Thus, despite the fact that techniques for testing of single-input programs such as fuzzing [39] and symbolic execution [7], [12], [19] have made impressive progress in recent years, their extensions to stateful protocols [25], [28], [29] have not yet achieved the same level of effectiveness as their single-input counterparts: these approaches are not able to consider a sufficiently large range of adversarial inputs, and may miss bugs that are exposed by specific input sequences.

Work partially funded by the Swedish Research Council (Vetenskapsrådet) and the Swedish Foundation for Strategic Research through project aSSIsT.

In this paper, we describe our experiences using symbolic execution to thoroughly test implementations of a network protocol against its specification. We employ a methodology, in which we first form a specification by extracting requirements from the protocol’s RFC and formulating them as assertions over the sequence of packets exchanged during a session. The extraction can be done incrementally, considering individual requirements separately. One such requirement could for example concern the sequence numbers in a sequence of packets, another one the version negotiation, etc. We then use symbolic execution to search for code paths and corresponding packet sequences that violate these requirements. In order to test the processing of a sequence of packets, each protocol party is tested separately: we consider the packets it receives as a sequence of inputs and check the party’s output when processing this sequence. We leverage symbolic execution to explore the code paths that an implementation may traverse when processing a packet sequence, tailoring it to explore only those code paths and inputs which expose potential requirement violations. Whenever such a violation, crash, or memory error is observed, the symbolic execution engine generates specific values that trigger this error, which are used to construct a complete test case to validate the bug. To achieve scalability for symbolic execution over a packet sequence, only those parts of input packets that are relevant for the tested requirement are made symbolic, using a pre-recorded sequence of packets as a basis.

We applied our methodology to test four implementations of DTLS against the protocol’s RFC. The DTLS (Datagram Transport Layer Security) protocol is a variation of TLS over UDP. DTLS is widely used in wireless networks, and is currently one of the primary protocols for securing IoT applications [30]. Two of the DTLS implementations we tested (OpenSSL and MbedTLS) are among the most widely used and well-tested. Using our methodology, we were able to detect numerous, previously unknown, security vulnerabilities and non-conformance issues in them, which have been confirmed and/or fixed by now.

In short, the main contributions of our work are:

- We describe an experience in using symbolic execution to thoroughly test implementations of a network protocol against its specification, in which we tailor symbolic execution to explore code paths that can expose violations of requirements formulated over the processing of a sequence of packets in a session.

- We describe how our effort was applied to four DTLS implementations against the protocol's RFC, demonstrating the effectiveness of our methodology on DTLS by revealing more than thirty previously unknown vulnerabilities and non-conformance bugs in well-tested DTLS implementations.
- We provide replication material [2] for all our experiments.

Outline: After an overview of our effort in the next section, in §III we show how requirements from DTLS's RFC are encoded as formulas. We describe our implementation in §IV, evaluate its effectiveness in §V, discuss related works in §VI, and end with some final remarks.

II. OVERVIEW OF OUR METHODOLOGY

In this section, we give a high-level overview of the methodology we used. It consists of three steps: i) a step where requirements are extracted from the protocol's RFC or specification and turned into formulas; ii) a step where (negations of) these formulas are represented as assumptions and assertions in the source code of the protocol's implementation, and where symbolic execution is used to explore code paths that expose bugs in the implementation and produce values for (selected) parts of the input(s) that follow these paths; and iii) a step where complete test cases for these bugs are constructed and are used to validate the bugs in the unmodified implementation. We detail these steps below.

A. Extracting Specification Requirements

Protocols encompass specific requirements over sequences of packets exchanged by two or more parties, sequences which are also known as *protocol sessions*. For DTLS, as well as for any other network protocol, we can extract a set of these requirements by scrutinizing its Request for Comments (RFC) [27, p. 13]. This task is facilitated by the fact that it is common for RFCs to use particular keywords (e.g., "MUST", "MUST NOT", "REQUIRED", "SHALL NOT", etc.) to signify the strictness of the protocol's requirements [5]. For example, the keyword "MUST" expresses that the definition is an absolute requirement of the specification, while "SHOULD" indicates a recommendation. Starting from these keywords, one can derive a set of requirements that can be represented by formulas over sequences of inputs and outputs.

A prominent class of requirements, which we will refer to as *input validity requirements*, concerns checking well-formedness of (the sequence of) inputs. For example, the following sentence from the DTLS RFC, specifies a requirement regarding uniqueness of record sequence numbers in sessions:

For each received record, the receiver MUST verify that the record contains a sequence number that does not duplicate the sequence number of any other record received during the life of this session.

For a set of records R received during a DTLS session, we can represent this requirement in predicate logic by the formula:

$$\forall r, r' \in R: r \neq r' \implies r.\text{sequence_number} \neq r'.\text{sequence_number} \quad (1)$$

Another class of requirements concerns output generated in response to (a sequence of) inputs. We refer to these as

input-output requirements. For example, the following sentence, extracted from the DTLS RFC [27, p. 17] and its errata (ID: 5186), concerns the sequence number in a server's response.

In order to avoid sequence number duplication in case of multiple cookie exchanges, the server MUST use the message_seq in the ClientHello as the message_seq in its initial ServerHello.

Letting $\text{resp}(r, i)$ represent the i -th output generated in response to input record r , we can express this requirement by the formula:

$$\begin{aligned} \forall r \in R: r.\text{msg_type} = \text{client_hello} \implies \\ (\text{resp}(r, 1).\text{msg_type} = \text{server_hello} \implies \\ \text{resp}(r, 1).\text{message_seq} = r.\text{message_seq}) \end{aligned} \quad (2)$$

which uses the fact that the `msg_type` field of a record r contains the type of its enclosed (and perhaps fragmented) message.

B. Symbolic Execution

Given a set of requirements expressed using formulas such as (1) and (2), our goal is to detect bugs (constraint violations, non-conformances, crashes and security vulnerabilities that may be associated with not adhering to the RFC, etc.) in protocol implementations (SUTs). Conceptually, the core idea of this step is simple. For each requirement we insert instrumentation into the SUT's source code as follows: 1) input constraints under which the requirement *can* be violated are inserted as *assumptions* on input to the SUT's source code, and 2) checks whether the SUT actually performs an action that violates the requirement are inserted as *assertions*. We then use *symbolic execution* to explore those executions that satisfy the assumptions on input, looking out for actions that trigger an assertion violation, crash, or memory error, and constructing test cases that are *witnesses* for each of these bugs.

Symbolic execution analyzes programs for which (some of) the input variables are designated as *symbolic*. It explores the code paths that are possible for some values of the symbolic input, and also provides input values that make executions follow specific code paths. In order to apply symbolic execution to a SUT, we represent the sequence of input packets as structured-type variables, which can be named after their message types as `client_hello`, `client_key_exchange`, etc. Making complete packets symbolic would in many cases not scale, therefore only the fields that can be expected to be relevant for the considered requirement are made symbolic. For instance, the payload, length, and several other fields, are unlikely to affect the processing of sequence numbers, and need not be made symbolic when checking requirements on sequence numbers. The non-symbolic fields must then be given concrete default values; we take these from the packet sequence in a pre-recorded session.

Let us illustrate the above with the examples from §II-A. An input validity constraint, such as Formula (1), refers to a typically unrestricted set R of records received during a DTLS session¹. We can specialize R to be a set of three records carrying message types `client_hello2`, `client_key_exchange`,

¹Occasionally, the RFC might specify related constraints that limit R .

and `change_cipher_spec`. In this set, all its elements have a `sequence_number` field. To prepare for symbolic execution, we pre-record a session containing these message types, represent the records as structured-type variables and make their `sequence_number` fields symbolic. We then add to the SUT's code the following assume statement (in C/C++ syntax), assuming the negation of the constraint expressed by that formula (as this is the constraint under which the requirement can be violated):

```
assume( ! // negate the conjunction below
(client_hello2.sequence_number != client_key_exchange.sequence_number &
client_hello2.sequence_number != change_cipher_spec.sequence_number &
client_key_exchange.sequence_number != change_cipher_spec.sequence_number))
```

What we have done here is to manually construct the pair-wise conjunction that the universal quantifier of Formula (1) will produce, by specializing it to the case of three records with the chosen message types. Of course, when the formulas are more complicated or the sets of records that they involve are larger, such assume statements can call appropriate auxiliary functions that we add to the SUT instead of specifying the constraints inline, as we did here. We must also add an assert statement to check that the protocol does not use invalid input in some forbidding way. For this case, the DTLS 1.2 RFC [27, p. 14] specifies that “*Invalid records SHOULD be silently discarded*”, so we may want to check whether an implementation achieves progress (e.g., by changing states) even after reception of invalid records. In implementations that already provide an API to inspect their internal state(s) or when it is easy to add one, the detection of such non-conformances can be fine-grained and very precise. A coarser-grained approach, which is the one we have implemented, is to simply check whether protocol interactions complete successfully (e.g., handshake is achieved) even in the presence of an invalid record in a test; this is done by an `assert(false)` at the point where the SUT is about to complete the interaction. For the DTLS protocol, this clearly indicates the violation of a “SHOULD”-type requirement, and thus non-conformance of the SUT to the standard.

For input-output requirements, such as Formula (2), we augment the SUT with structure-type variables that represent relevant output packets. Specializing `resp(r, i)` to the case where `r` is the packet `client_hello2`, we add the assertion

```
assert( // checks message_seq validity
(resp(client_hello2,1).msg_type != server_hello) |
(resp(client_hello2,1).message_seq == client_hello2.message_seq))
```

Since an input-output requirement can be violated only on valid inputs, we add an assume statement ensuring the validity of the field `client_hello2.message_seq`.

In summary, a protocol requirement expressed by a first-order logic formula is checked by augmenting the SUT as follows: 1) variables are inserted to represent the input packets and (in the case of input-output requirements) relevant output packets; 2) the formula is specialized by instantiating its quantifiers to the possible values in the pre-recorded sequence, producing a quantifier-free expression over the packet-representing variables; 3) for an input validity requirement, its negation is added in an assume statement, and an assert statement is added to check that the invalid inputs are not handled in a forbidden way;

4) for an input-output requirement, the expression is added in an assert statement checking correctness of the output, together with an assume statement ensuring the validity of inputs.

C. Test Case Construction and Validation

Starting from some pre-recorded valid input sequence S of records as seed, and given some time budget T , a symbolic execution tool such as KLEE [7] will explore a number of code paths of SUT^a (the SUT that has been extended with assume and assert statements and where some fields have been made symbolic). For each code path that satisfies the assume statements and triggers an assertion violation, crash, or memory error, we record and return a tuple of values $\langle v_1, \dots, v_k \rangle$ for the fields f_1, \dots, f_k of records in S which have been made symbolic. For each such code path, we can therefore construct a test case by substituting the value of record field f_i with v_i and keeping all other values in records of seed S unchanged. Each of these test cases executes a unique code path of SUT^a. We can simply run these test cases in the original SUT for validation. For tests that result in crashes (e.g., segmentation faults, memory errors, etc.), the existence of a bug in the SUT is clear. Tests that trigger assertion violations expose non-conformances and other policy violations, given our instrumentation to check protocol requirements.

III. DTLS AND ITS SPECIFICATION REQUIREMENTS

DTLS is a client-server protocol that secures communication over datagram-based transport layer protocols. It is an adaptation of TLS, fulfilling a similar purpose, but doing so for unreliable transport layer protocols such as UDP. DTLS is available in two standardized versions, 1.0 [26] and 1.2 [27], which are respectively based on TLS 1.1 [14] and TLS 1.2 [15].

DTLS secures communication through the establishment and use of session keys. To that end, it is structured in several layers. The Record layer performs encryption and encapsulation of sent data and decryption and decapsulation of received data. The Handshake layer establishes session keys and the cryptographic algorithms to use at the Record layer. Other layers include ChangeCipherSpec that handles key deployment, Application, which provides a carrier for application data, and Alert, which is used to signal unexpected events. Our work concentrates on the Record and Handshake layers. We first explain the operation of these layers, before introducing the requirements defined for each layer.

The Record layer sits at the base, with all other layers operating directly on top of it. It splits received network data into *records*. The payload of these records is decrypted and authenticated using session keys. The decrypted payload is divided into *messages*, which are dispatched to the corresponding upper layer. In the opposite direction, the Record layer encrypts messages received from upper layers, encapsulates them into records, and packs the records inside datagrams for the transport layer. Before deployment of session keys, the Record layer operates without encryption and authentication (e.g., messages are extracted directly from record payloads).

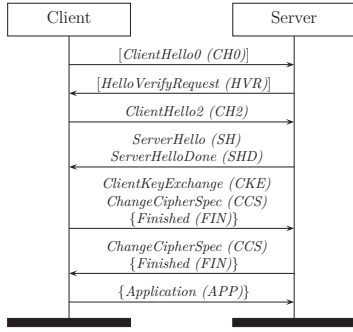


Fig. 1: DTLS handshake when pre-shared key (PSK) is used. Encrypted messages are inside braces. Optional messages are inside square brackets.

The Handshake layer is engaged at the start of each DTLS session in order to establish session keys and the cryptographic algorithms to use at the Record layer. This is achieved by completion of a DTLS *handshake*. Over the course of this handshake, the client and server exchange flights of handshake messages in a specified order; see Fig. 1 showing a handshake using the pre-shared key (PSK) algorithm. The exact form of the handshake varies depending on the key exchange algorithm used to generate session keys. In addition to key establishment, the Handshake layer is also tasked with fragmentation and reassembly of handshake messages. Handshake messages when packed inside a record may be too large to fit inside a datagram. Consequently, these messages are first split by the Handshake layer into *fragments*, before being passed on to the Record layer. Conversely, fragments received from the Record layer are first assembled to form *unfragmented messages*, and only then processed. We shall revisit these terms in §III-B.

A. Record Layer Requirements

A DTLS record is a structured entity whose fields and types are specified in the DTLS RFC [27, p.6]. Fields which are DTLS-specific (i.e., not part of the corresponding TLS record) are identified with the comment *New field*. As we can see, the epoch number field is specified as a 16-bit unsigned integer, and the record sequence_number as a 48-bit unsigned integer. The type and version fields are defined via *enumerated types*. The value of the type field determines the upper layer the message is destined for. For example, if the value of type is handshake, the fragment should be passed on to the Handshake layer, etc. Finally, the fragment field, denoting the message contained in the record, is defined as a one-dimensional vector.

```

struct {
  ContentType type;
  ProtocolVersion version;
  uint16 epoch; // New field
  uint48 sequence_number; // New field
  uint16 length;
  opaque fragment[DTLSPlaintext.length];
} DTLSPlaintext;

```

Given this structure and the TLS and DTLS RFCs, we can formulate constraints over fields of a set of records R used in a DTLS session, as follows.

1) *Record Version*: The DTLS RFC [27, p.4] specifies only two versions for DTLS. The formula for this field is simple:

$$\forall r \in R: r.version \in \{\text{DTLS1.0}, \text{DTLS1.2}\}$$

2) *Record Length*: The TLS 1.2 RFC [15, p.20] defines record length as: “The length (in bytes) of the following *TLSPlaintext.fragment*.” Moreover, the DTLS 1.2 RFC [27, p.8] specifies that the length is “Identical to the length field in a TLS 1.2 record.” Hence, the length field in DTLS represents the number of bytes stored in the DTLS record’s fragment field. We capture these requirements in the following formula, where *num_bytes* is a function returning the size of its argument (in bytes).

$$\forall r \in R: r.length = \text{num_bytes}(r.fragment)$$

So far, the requirements for record fields were simple and quantifying over the set of records R with variables r, r', \dots sufficed. But there are also RFC requirements that specify relations between consecutive records or even sequences of records that a protocol party receives during a session. We then equip r with a subscript that denotes the record’s position in the sequence. Thus r_1, r_2, \dots is the sequence of records in a session (r_1 is the first record, r_2 the second, etc.). We can also quantify over subscripts. As well, for a record r_i , its previous record is obtained as r_{i-1} and its next as r_{i+1} .

3) *Epoch Number*: The epoch field is a two-byte unsigned integer used to distinguish the session keys a record was encrypted with. The following excerpt from RFC [27, p.8] explains how this field should be updated.

The epoch should be initialized with zero, and should be incremented with every ChangeCipherSpec message sent.

For n received records in a DTLS session, we capture this requirement with the following formula:

$$\forall i \in \mathbb{N}, 2 \leq i \leq n: \\ (r_1.epoch = 0) \wedge \\ (\text{if } r_{i-1}.type = \text{change_cipher_spec} \text{ then } r_i.epoch = r_{i-1}.epoch + 1 \\ \text{else } r_i.epoch = r_{i-1}.epoch)$$

This formula specifies that the epoch of the first record is zero and that for all subsequent records the value of the epoch field is either one more than the value of the previous record or has the same value as it, depending on whether the value of the type field of the previous record is *change_cipher_spec* or not.

4) *Record Sequence Number*: The DTLS 1.2 RFC [27] defines the record sequence_number as a 48-bit unsigned integer for the purpose of preventing replay attacks. Section II-A quoted a requirement regarding the uniqueness of the sequence_number and showed a formula for this requirement (Formula 1). In the excerpt below, the RFC [27, p.13,14] additionally requires rejection of records whose sequence_number values are smaller than the lowest sequence_number in the current window (i.e., the value at the left edge of the window).

Duplicates are rejected through the use of a sliding receive window. A window size of 64 is preferred and SHOULD be employed as the default. The "right" edge of the window represents the highest validated sequence number value received on this session. Records that contain sequence numbers lower than the "left" edge of the window are rejected. If the received record falls within the window and is new, or if the packet is to the right of the window, then the receiver proceeds to MAC verification.

Assuming a max function that returns the maximum of a set of values, the requirements for sequence number values are given by the following formula:

$$\forall i, j \in \mathbb{N}, 1 \leq i, j \leq n, i \neq j: \\ (r_i.\text{sequence_number} \neq r_j.\text{sequence_number}) \wedge \\ (\max(\{r_k.\text{sequence_number} - 64 \mid 1 \leq k < i\} \cup \{0\}) \leq r_i.\text{sequence_number})$$

The first conjunct is the uniqueness requirement; the second conjunct specifies the “windowing” requirement.

B. Handshake Layer Requirements

The Handshake layer exchanges a sequence of messages to establish session keys and cryptographic algorithms. Larger handshake messages may be split into fragments. An unfragmented message is also regarded as a fragment consisting of an entire message. The Handshake layer passes to the Record layer a sequence of fragments, which are encrypted if session keys have been deployed, and then transmitted as the `fragment` fields in a sequence of records. Handshake fragments have the structure shown below. The fields `msg_type` and `length` indicate the type and length of the (unfragmented) message from which this fragment was derived, whereas `message_seq` indicates the order, starting from 0, of that message in the handshake interaction. The body field stores a fragment of the original message, whose structure varies depending on `msg_type`. The offset and the length of this fragment are given by the `fragment_offset` and `fragment_length` fields, respectively.

```
struct {
    HandshakeType msg_type;
    uint24 length;
    uint16 message_seq;
    uint24 fragment_offset;
    uint24 fragment_length;
    select (HandshakeType) {
        case hello_request: HelloRequest;
        case client_hello: ClientHello;
        // eight more cases here...
        case finished: Finished;
    } body;
} Handshake;
```

For the Handshake layer, we define formulas over a set M of received messages, using m to range over individual messages.

1) *Handshake Type*: The RFC [27, p. 27] specifies the valid values for `msg_type` as belonging to the following enumeration:

```
enum HandshakeType {
    hello_request = 0, client_hello, server_hello, hello_verify_request,
    certificate = 11, server_key_exchange, certificate_request,
    server_hello_done, certificate_verify, client_key_exchange,
    finished = 20 };
```

We can capture the requirement that the valid values for `msg_type` are those in the enumeration above with the formula:

$$\forall m \in M: m.\text{msg_type} \in \{\text{hello_request}, \text{client_hello}, \dots, \text{finished}\}$$

We can strengthen this formula if we additionally consider the ordering of messages imposed by the RFC [27, p. 21]. This allows us to capture a requirement enforcing correspondence between the order of a message (given by `message_seq`) and its specified `msg_type`. Formulating this requirement is quite demanding: the order of a message for a given type may vary depending on the key exchange the handshake uses, and on the side receiving the messages (whether it is a client or a server). To keep the presentation concise, we show the formula for a handshake that uses PSK (taking the form shown in Fig. 1)

and the side in question is a server. For this particular case, the requirement can be formulated as follows:

$$\forall m \in \text{serverPSK}(M): \\ (m.\text{message_seq} = 0 \implies m.\text{msg_type} = \text{client_hello}) \wedge \\ (m.\text{message_seq} = 1 \implies m.\text{msg_type} = \text{client_hello}) \wedge \\ (m.\text{message_seq} = 2 \implies m.\text{msg_type} = \text{client_key_exchange}) \wedge \\ (m.\text{message_seq} = 3 \implies m.\text{msg_type} = \text{finished})$$

The function `serverPSK` returns the messages from M which the server receives during PSK handshakes. Similar requirements can be formulated for clients, as well as for handshakes using other key exchange algorithms.

Similarly as for records, there are RFC requirements that specify relations between consecutive messages. We equip m with a subscript that denotes the message’s position in the sequence. Thus m_1, m_2, \dots is the sequence of received messages during a session. We can also quantify over subscripts.

2) *Message Sequence*: The RFC [27, p. 18,19] specifies the following requirement for `message_seq`:

The first message each side transmits in each handshake always has message_seq = 0. Whenever each new message is generated, the message_seq value is incremented by one. When a message is retransmitted, the same message_seq value is used.

The excerpt suggests that the value for `message_seq` equals zero in the initial message and is incremented with each following message, except when the same message is retransmitted, in which case `message_seq` is unchanged. To capture this requirement, we must characterize when a message is “the same” as the previous. We approximate this by equality of their body fields. This leads to the following formula:

$$\forall i \in \mathbb{N}, 2 \leq i \leq n: \\ (m_1.\text{message_seq} = 0) \wedge \\ (\text{if } m_i.\text{body} \neq m_{i-1}.\text{body} \text{ then } m_i.\text{message_seq} = m_{i-1}.\text{message_seq} + 1 \\ \text{else } m_i.\text{message_seq} = m_{i-1}.\text{message_seq})$$

In order to capture requirements related to fragmentation of messages, we continue to assume a sequence m_1, m_2, \dots of messages, and additionally assume that message m_i is fragmented into `num_fragments(i)` fragments. We use $m_i[j]$ to refer to fragment j of message i . The next four requirements concern fragmentation.

3) *Fragment Reassembly*: For reassembling (un)fragmented handshake messages, the RFC [27, p. 20] mandates that:

When a DTLS implementation receives a handshake message fragment, it MUST buffer it until it has the entire message.

In other words, this RFC excerpt requires that every byte of the original message should exist in one of the fragments. We capture the requirement with the aid of `fragment_offset` and `fragment_length` fields in the following formula:

$$\forall i \in \mathbb{N}, 1 \leq i \leq n, \forall b, 0 \leq b < m_i.\text{length}, \exists j, 1 \leq j \leq \text{num_fragments}(i): \\ m_i[j].\text{fragment_offset} \leq b < m_i[j].\text{fragment_offset} + m_i[j].\text{fragment_length}$$

4) *Unfragmented Message Offset / Length*: If a handshake message fits in a datagram, fragmentation will not take place. In this case, the RFC [27, p. 20] specifies the following requirement on `fragment_offset` and `fragment_length`:

An unfragmented message is a degenerate case with $fragment_offset = 0$ and $fragment_length = length$.

With the help of the `num_fragments` function, this requirement can be formulated as follows:

$$\forall i \in \mathbb{N}, 1 \leq i \leq n: num_fragments(i) = 1 \implies (m_i.fragment_length = m_i.length) \wedge (m_i.fragment_offset = 0)$$

5) *Message Sequence in Fragments*: In describing fragmentation, the RFC [27, p. 20] specifies the following requirement:

The sender then creates N handshake messages, all with the same `message_sequence` value as the original handshake message.

We can infer that the `message_sequence` of all fragments should be equal to the `message_sequence` of the unfragmented message. We can capture this requirement using the following formula:

$$\forall i \in \mathbb{N}, 1 \leq i \leq n, \forall j, 1 \leq j \leq num_fragments(i): m_i[j].message_sequence = m_i.message_sequence$$

6) *Message Length in Fragments*: The RFC [27, p. 20] makes a similar remark regarding the `message_length`:

The length field in all messages is the same as the length field of the original message.

Similarly, we can formulate this requirement as the following:

$$\forall i \in \mathbb{N}, 1 \leq i \leq n, \forall j, 1 \leq j \leq num_fragments(i): m_i[j].length = m_i.length$$

7) *Handshake Version*: Similar to how records of the DTLS Record layer contain the DTLS version, there are version fields in the message bodies of *ClientHello*, *HelloVerifyRequest* and *ServerHello* messages. Similar to III-A1, we can formulate a requirement for `client_version` and `server_version`.

8) *Requirements on Length of Fields*: Several handshake messages contain fields that define the length (in bytes) of other fields. Such fields occur in the body of a handshake message (we do not show its structure here). We can capture requirements for such fields in a similar way as for Record Length formula in §III-A2. For space reasons, we omit their formulas.

We end this section by mentioning that we have formulated more input validity requirements for DTLS; we chose to present only the ones above because they revealed bugs in the implementations we tested. In our evaluation (§V), we will also check two input-output requirements; that of Formula (2) and another one which is similar. We will refer to these requirements as *CH0/HVR* and *CH2/SH Message Sequence*.

IV. IMPLEMENTATION

In this section, we describe our implementation, including efforts to overcome some obstacles. Our implementation is structured as follows. First, we prepare the SUT’s code for symbolic execution (§IV-A). Second, we capture concrete records of a handshake interaction (§IV-B). For a given requirement, we then make symbolic the fields of captured records that participate in it and insert corresponding assertions and assumptions (§IV-C). The functionality for doing so is provided in a shared library, based on which a series of test harnesses were developed, one for each SUT we tested. The SUT is executed symbolically using KLEE [7]. We then

construct test cases (witnesses) from the values of symbolic variables that KLEE generates when requirement-violating behaviors or crashes are detected. These test cases are used to check if the problems can be reproduced on the unmodified SUT (§IV-D). We detail these steps below.

A. SUT Preparation

To prepare the SUT for symbolic execution, we first make modifications to its code to ensure it executes deterministically; this is necessary to make it respond in the same way during record capture as during the subsequent symbolic execution. This involves de-randomization (e.g., in TinyDTLS we modified the function `dtls_fill_random`, used to compute client/server random nonces, to fill a buffer with 1’s instead of a random value), and disabling retransmissions. Finding the code places where such changes should be made typically involves a search for the ‘random’ keyword; the changes themselves are simple.

As mentioned in §II-B, we use completing a DTLS handshake as a means of determining whether input validity requirements are violated. To detect when the handshake is near completion, we insert an `assert(false)` at the point where the SUT is expecting *Finished*. KLEE will terminate the current path and generate a corresponding sequence of input values upon executing such a statement. One may wonder why failing assertions are inserted when *Finished* is expected and not after the handshake is completed. The main reason is to steer symbolic execution away from complicated code used for decryption and authentication. (Recall from Fig. 1 that *Finished* is the first encrypted message that a side receives.) This does mean, however, that in order to validate the bug in the unmodified SUT, we have to provide a valid *Finished* message outside of symbolic execution. We revisit this matter in §IV-D.

B. Record Capture

The next step is to capture the records of a DTLS handshake. We opted for a PSK handshake of the form shown in Fig. 1, which by design omits certain cryptographic operations (e.g., such as those in authentication and key exchange) compared to handshakes using other key exchange algorithms. This minimizes the amount of cryptographic code that is executed symbolically, and also makes symbolic execution faster.

To capture records, we designed a DTLS test program that starts by instantiating a client and a server. Upon instantiation, the client generates an initial *ClientHello* record. Records are then passed back and forth between the two parties, with each transmitted record also stored in a separate file. The end result is a folder containing a file for each record in the handshake.

The client/server interaction is performed over the SUT’s API. However, DTLS does not have a standardized API. Consequently, the methods for initialization and sending/receiving record data differ between the SUTs. This made coding the initial test program time-consuming (i.e., in the range of a few days) since it required familiarization with each SUT’s API. We got inspiration from existing demo programs, in particular from a program used to fuzz the Mbed TLS library [18].

C. Symbolic Execution

The test program that captures records is extended to a *test harness* for symbolic execution with the aid of the *shared library* developed as part of this work. The shared library contains three types of functions:

- 1) helper functions (e.g., for loading the records from file),
- 2) functions to parse DTLS records into pre-defined DTLS data structures with a structure closely resembling that of DTLS records presented in §III-A, and
- 3) functions to make specific fields of records symbolic, and for forming boolean expressions in `assumes` and `asserts`.

By defining these functions in a shared library, we can reuse their functionality to test different DTLS implementations. This is done by linking the shared library to a new implementation and simply calling the functions from the test harness. Minor adaptations are needed to reflect any new extensions a SUT may support (present in the *ClientHello* and *ServerHello* extension headers) or different lengths of the variable-length fields (primarily for the `cipher_suites` field in *ClientHello*).

Given a requirement, the test harness is used to symbolically execute one side (a client or a server). It operates as follows.

Record Loading: The harness first calls the shared library to load the records received by that side during the captured handshake and stores them in separate data structures. For example, function `parse_CH0(uint8_t *buf, CH0 *rec)` initializes a *ClientHello0*-structured record with data from a buffer.

Making Fields Symbolic: Fields that appear in a requirement are made symbolic using `klee_make_symbolic`. Thereafter, the assumptions on input in the formalization are inserted as `klee_assume` statements, and the assertions on output as `klee_assert` statements. To aid this step, our shared library provides methods for constructing the relevant boolean expressions. For example, it includes the method:

```
void epochServer(CH0 *client_hello0, CH2 *client_hello2,  
                CKE *client_key_exchange, CCS *change_cipher_spec);
```

which makes the `epoch` fields in all supplied record entities symbolic, and assumes the negation of the epoch number constraint defined in §III-A3. Note that we do not make fields of *Finished* records symbolic for performance reasons similar to those given in §IV-A. This step ends by executing the test harness symbolically using KLEE.

D. Test Case Construction and Validation

During symbolic execution, for each path explored, KLEE generates corresponding values for the symbolic variables. These values are stored in separate files, with sequences leading to errors named differently, making them easily distinguishable. From these values and captured records, we build corresponding test input records which we use to validate that the problem exists (i.e., the handshake is completed) in the unmodified SUT. Validation is done by supplying test input records over UDP to client/server utilities provided with the implementation. For most requirements, validation can be performed using the input records generated via symbolic execution on a de-randomized SUT without assertions and assumptions. This exposes bugs

in the same way as on the unmodified SUT, since the de-randomization does not affect control flow. One exception to when validation can be performed in this way is when it requires some input records (e.g., *Finished*) to be constructed from previous ones using cryptographic functions. For these cases, we use existing DTLS libraries (TLS-Attacker [34] and DTLS-Fuzzer [17]) to generate test cases which expose the requirement violation.

V. EVALUATION

In parallel with extracting requirements for DTLS based on its RFC and its errata, we looked for previously reported bugs for DTLS implementations. We came across CVE-2014-0195, an exploitable security vulnerability of OpenSSL, which is explained in detail in this blog [38]. The bug is in OpenSSL's function for reassembling fragmented messages, and prompted us to implement the Handshake layer requirements regarding fragmentation (§III-B3 to §III-B6). One of the latest versions with this vulnerability is OpenSSL 1.0.1f. So we took its code and tested it. We were able to expose the vulnerability very quickly. When using OpenSSL as a server, the bug, which violates the Message Length requirement (§III-B6), was detected in just 20 seconds, and when using OpenSSL as a client in 38 seconds. In both cases, KLEE version 2.2 explored just eight paths in OpenSSL's code. Out of curiosity, we also checked the code of OpenSSL 1.0.1f for other problems, and discovered, again in less than 20 seconds, two minor non-conformance issues in its code. Both issues were previously unknown and concern not checking the DTLS version requirements (§III-A1 and §III-B7). This warm-up experiment, and in particular exposing a (known but serious) security vulnerability without much effort and very quickly, convinced us of the power and promise of our methodology. So we were eager to find out what we could discover in newer, and hopefully more robust, DTLS implementations. We present these results in the remainder of this section.

SUTs: For our evaluation, we chose a total of four different DTLS implementations. Two of them, *OpenSSL* and *Mbed TLS*, are well-known and widely used. For OpenSSL, we used *3.0.0-alpha12*, the most recent pre-release of version 3.0.0 at the time of our evaluation. For Mbed TLS, we used version *2.22.0*. The other two implementations are variants of *TinyDTLS*, a lightweight DTLS implementation targeting IoT devices. The first variant, which we will denote as *TinyDTLS^E*, is hosted and maintained by the Eclipse IoT project. The second variant, which we will denote as *TinyDTLS^C*, was branched out from Eclipse's IoT project, developed independently ever since, and is used in the Contiki-NG operating system. For both *TinyDTLS* variants, we used the latest commits of their development branches (*7068882* and *53a0d97*, respectively) at the time our evaluation was conducted. Because the 'develop' branch of Eclipse's *TinyDTLS* does not support Handshake layer fragmentation, we used the most recent commit (*94205ff*) of its 'handshake_fragmentation' branch to test *TinyDTLS^E*'s support for DTLS's handshake fragmentation requirements.

TABLE I: Number and classes of bugs found in the SUTs we used.

	OpenSSL		Mbed TLS	TinyDTLS ^E		TinyDTLS ^C
	1.0.1f	3.0.0-alpha12	2.2.2.0	7068882	94205ff	53a0d97
Vulnerability	1	1	–	3	+0	2
Other	–	–	–	3	+1	1
Non-conformance	2	2	3	9	+1	10

Summary of Results: Table I shows a breakdown of the total of 36 bugs (not counting the three in OpenSSL 1.0.1f) that we detected across the five SUTs that we consider in our evaluation. In the table, we use the notation $+N$ for the bugs in the handshake fragmentation branch of TinyDTLS^E to refer to the additional bugs that this branch contains compared to the develop branch (and so that we do not count the same bugs twice). All these bugs have been reported to the developers of these DTLS implementations, and most have been confirmed and corrected by now. Also, note that we detected bugs in all the DTLS implementations we considered. In Table I, we classify these bugs into vulnerabilities, non-conformance issues, and use “Other” for bugs that do not fall clearly into the other two classes.

In the next two subsections, we describe the bugs that we detected in OpenSSL and in Eclipse’s TinyDTLS implementation, and provide measurements about our experiments. We omit detailed descriptions of the three non-conformance bugs detected in Mbed TLS, and of the thirteen bugs detected in TinyDTLS^C because they are similar to those in TinyDTLS^E.

A. Bugs in OpenSSL

On OpenSSL 3.0.0-alpha12, we detected a serious vulnerability and a non-conformance bug (cf. Table I).

The vulnerability, which involves accessing an out-of-bounds pointer and crashing the SUT, is detected in less than two minutes both when using OpenSSL as a server and as a client (cf. Tables II and III), with the help of the Handshake Type requirement (§III-B1). The vulnerability has been reported to the OpenSSL developers as issue 14906 and quickly fixed. It occurs as follows: Under normal handshake, the client initiates a handshake by sending a *ClientHello* (*CH0*) message to the server. To trigger the vulnerability, the (malicious) client instead starts the handshake by sending a *CH0* message which is however tagged with `msg_type = finished`. This fools the server, which mistakenly tries to process a *Finished* message and access the non-existing at this point cipher suite elements (which would have been agreed by the client and the server if the handshake had been properly done and the *Finished* message would have been processed when its time had come). This causes the server to crash. In a similar scenario, during a handshake, if the server responds to a *ClientHello* message with a *Finished* message, the client crashes for the same reason. This is what the OpenSSL developers said about this bug: “Good catch! Fortunately this only affects the dev branch and not 1.1.1 (otherwise this would have been a CVE).” We mention in passing that the fix for this issue triggered a general discussion among developers about revisiting how OpenSSL 3.0.0 manages transcript hashes and that its state machine should be redesigned.

TABLE II: Results using the OpenSSL 3.0.0-alpha12 server instance as SUT.

Requirement	Bug	Paths	Time	Participating Records
Record Version	✗	4	1m19s	<i>CH0, CH2, CKE, CCS</i>
Record Length		492	1h56m01s	<i>CH0, CH2, CKE, CCS</i>
Epoch Number		10	2m02s	<i>CH0, CH2, CKE, CCS</i>
Record Sequence Number		74	21m05s	<i>CH2, CKE, CCS</i>
Handshake Type	✗	6	1m10s	<i>CH0, CH2, CKE</i>
Message Sequence		8	1m18s	<i>CH0, CH2, CKE</i>
Fragment Reassembly		51	4m46s	<i>CKE</i>
Unfragmented Message Offset		3	59s	<i>CH0, CH2, CKE</i>
Unfragmented Message Length	?	5579	⊙	<i>CH0, CH2, CKE</i>
Message Sequence in Fragments		16	1m56s	<i>CKE</i>
Handshake Version	✗	8	1h18m36s	<i>CH0, CH2</i>
Fragment Length		272	31m43s	<i>CH0, CH2, CKE</i>
Cookie Length		182	40m13s	<i>CH0, CH2</i>
Session ID Length		646	1h13m20s	<i>CH0, CH2</i>
CH0/HVR Message Sequence		3	1m04s	<i>HVR</i>
CH2/SH Message Sequence		4	1m13s	<i>SH</i>

TABLE III: Results using OpenSSL 3.0.0-alpha12 client instance as SUT.

Requirement	Bug	Paths	Time	Participating Records
Record Version	✗	5	1m47s	<i>HVR, SH, SHD, CCS</i>
Record Length		259	29m18s	<i>HVR, SH, SHD, CCS</i>
Epoch Number		10	2m28s	<i>HVR, SH, SHD, CCS</i>
Record Sequence Number		210	1h23m41s	<i>HVR, SH, SHD, CCS</i>
Handshake Type	✗	18	1m58s	<i>HVR, SH, SHD</i>
Message Sequence		19	1m58s	<i>HVR, SH, SHD</i>
Fragment Reassembly		1076	2h22m21s	<i>SH</i>
Unfragmented Message Offset		3	1m03s	<i>HVR, SH, SHD</i>
Unfragmented Message Length	?	5499	⊙	<i>HVR, SH, SHD</i>
Message Sequence in Fragments		23	2m26s	<i>SH</i>
Handshake Version	✗	6	43m49s	<i>HVR, SH</i>
Fragment Length		112	10m28s	<i>HVR, SH, SHD</i>
Cookie Length		21	6m34s	<i>HVR</i>
Session ID Length		3	1m00s	<i>SH</i>

This provides some evidence that the bugs exposed by our methodology are often quite deep.

Besides this security bug, we also detected two non-conformance bugs. The Record Version and the Handshake Version requirements (§III-A1 and §III-B7) of the RFC revealed that OpenSSL did not properly check for the `version` fields when the underlined records in Tables II and III were processed.

Let us also describe the information in the tables of this and the next subsection. Each of their rows shows the requirement which is checked, whether it revealed a vulnerability (✗), a non-conformance bug (✗) or was inconclusive (?) due to timeout (⊙), the number of different paths that KLEE explored and the time this required, and the shorthand for the participating records of the DTLS protocol that the requirement involves. Records that expose the bug(s) that were detected are shown in red color (for vulnerabilities) or underlined (for non-conformances). Regarding the time that symbolic execution requires, we notice that most requirements are checked quite fast (in a few minutes when testing OpenSSL 3.0.0-alpha12), but there also exist requirements where KLEE needs to examine a significant number of paths of the SUT and the tests take more than one to two hours to complete or even time out after running for a day. We also note that the time that checking the requirements requires is *not* proportional to the number of execution paths that KLEE explores; for example, notice the Handshake Version rows in Tables II and III and contrast them with other rows in these two tables. The Handshake Version experiment takes

considerably longer despite involving a relatively small number of paths. This is due to the fact that unlike other tested implementations, OpenSSL by default, computes the master secret using a hash over the contents of prior messages. These contents include symbolic fields (`client_version` in *CH0* and *CH2*), causing KLEE to symbolically execute complicated hash code. On a general note, the experiments on OpenSSL 3.0.0-alpha12 were by far the most time-consuming ones among the DTLS implementations we tested.

B. Bugs in TinyDTLS

On the two TinyDTLS^E branches we tested, we detected a total of seventeen bugs. We omit the details of the ten non-conformance bugs, and describe only the three vulnerabilities and the four “Other” errors (cf. Table I).

The vulnerabilities we detected were reported as issues 59, 69, and 74. In short, they are as follows: (1) When a client receives a *ServerHello* (*SH*) where the `length` field has a value smaller than the actual size of the record, the variable that represents the record size wraps around. Later this variable is used to read the extensions contained in the *SH*, causing the client to crash. (2) When a malformed *ClientHello0* with `length = 0` is sent to the server, a size variable used when the server is generating the cookie wraps around. This causes the server to try to compute an HMAC over a large portion of memory using the size variable, which in turn crashes the server. (3) If a *ClientHello2* message has an invalid value for its `version` field, the server fails to retrieve the cookie from the message. A flaw in the handling of this failure, in turn, causes the server to proceed with the handshake. This makes TinyDTLS servers susceptible to Denial-of-Service attacks.

The four “Other” bugs are also quite serious, but we do not classify them as vulnerabilities because their consequences are unclear; e.g., they do not cause crashes. They have been reported as issues 54, 55, 57 and 70. In short, we have detected: (1) A memory over-read when the client or the server is reassembling a fragmented message, occurring if the `fragment_length` is smaller than the size of the actual fragment. (2) A client accessing an invalid memory past the boundary of the fragment when the server sends a fragmented *ServerHello* to the client where some of the handshake fields are not present in the fragment. (3) An over-shift taking place when a client or a server receives a record with a `sequence_number` greater than 32. (4) Another memory over-read when a *HelloVerifyRequest* contains a 16-byte cookie, but `cookie_length` is greater than 16.

It is interesting to point out that, for some of these bugs, it took up to three pull request attempts before the root of the problem was identified and fixed. Each time, we were able to trigger the bug in another way and show to the developers that their fixes were insufficient. The fact that we had an automatic way to test the version of TinyDTLS^E with the proposed fix was very handy. Finally, note that testing TinyDTLS is very fast for all but one requirement (Record Length); cf. the times in Tables IV and V.

TABLE IV: Results using TinyDTLS^E server instance as SUT.

Requirement	Bug	Paths	Time	Participating Records
Record Version		15	6s	<i>CH0, CH2, CKE, CCS</i>
Record Length		33	19s	<i>CH0, CH2, CKE, CCS</i>
Epoch Number	✗	11	4s	<i>CH0, CH2, CKE, CCS</i>
Record Sequence Number	✗/✗	12	3s	<i>CH2, CKE, CCS</i>
Handshake Type		35	14s	<i>CH0, CH2, CKE</i>
Message Sequence	✗	11	11s	<i>CH0, CH2, CKE</i>
Fragment Reassembly	✗	300	1m05s	<i>CKE</i>
Unfragmented Message Offset	✗	1	5s	<i>CH0, CH2, CKE</i>
Unfragmented Message Length	✗/✗	7	7s	<i>CH0, CH2, CKE</i>
Message Sequence in Fragments	✗	4	2s	<i>CKE</i>
Handshake Version	✗	2	12s	<i>CH0, CH2</i>
Fragment Length	✗/✗	7	4s	<i>CH0, CH2, CKE</i>
Cookie Length	✗	6	8s	<i>CH0, CH2</i>
Session ID Length	✗	32	16s	<i>CH0, CH2</i>
CH0/HVR Message Sequence	✗	2	7s	<i>HVR</i>
CH2/SH Message Sequence	✗	3	9s	<i>SH</i>

TABLE V: Results using TinyDTLS^E client instance as SUT.

Requirement	Bug	Paths	Time	Participating Records
Record Version		65	34s	<i>HVR, SH, SHD, CCS</i>
Record Length	✗	112	2h06m13s	<i>HVR, SH, SHD, CCS</i>
Epoch Number		21	6s	<i>HVR, SH, SHD, CCS</i>
Record Sequence Number	✗/✗	146	58s	<i>HVR, SH, SHD, CCS</i>
Handshake Type	✗	256	1m36s	<i>HVR, SH, SHD</i>
Message Sequence	✗	23	15s	<i>HVR, SH, SHD</i>
Fragment Reassembly	✗	726	2m55s	<i>SH</i>
Unfragmented Message Offset	✗	1	2s	<i>HVR, SH, SHD</i>
Unfragmented Message Length	✗	1	2s	<i>HVR, SH, SHD</i>
Message Sequence in Fragments	✗	4	2s	<i>SH</i>
Handshake Version	✗	2	3s	<i>HVR, SH</i>
Fragment Length	✗	1	2s	<i>HVR, SH, SHD</i>
Cookie Length	✗	18	19s	<i>HVR</i>
Session ID Length	✗	4	7s	<i>SH</i>

VI. RELATED WORK

Symbolic execution [19] was introduced already in the 70’s. In the last 15 years, improvements in tool implementations [7], [12] have made symbolic execution a powerful software testing technique. For testing network protocols, several works (e.g., KleeNet [29], SymNet [28], etc.), search for issues arising during the joint execution of several protocol parties combined with a test environment. Symbolic execution is employed to explore as many code paths as possible within some time budget. Since each input is generated by another party, these approaches mainly detect interoperability issues, and have limited power to detect flaws in processing of adversarial (non-valid) inputs.

SYMBEXNET [32] jointly executes the protocol parties symbolically to generate test inputs that explore as many code paths as possible in a given time budget. The generated test inputs are then replayed against the SUT, while monitoring its behavior to check whether any requirement extracted from the specification is violated. In contrast to our approach, SYMBEXNET symbolically executes the SUT without first augmenting it with assumptions or assertions that check for requirement violations. As a consequence, it may miss violations corresponding to paths which are exercised by both violating and non-violating inputs. To understand why, consider the Handshake Version requirement (§III-B7), which constrains the `client_version` field to a value in the set {DTLS1.0, DTLS1.2}. MbedTLS checks this requirement incorrectly;

it checks whether `client_version` is in a range which also includes the invalid version `DTLS1.1`. As a result, the same code path can be exercised by both valid and invalid version fields. For such a path, symbolic execution without inserted assumptions will generate a single test case which may or may not expose the bug (i.e. it may or may not have `client_version` set to `DTLS1.1`). In contrast, the assumptions used in our approach specifically address this, guiding symbolic execution to generate test cases that expose violations whenever possible. Similar examples include the missing checks for Message Offset/Length (§III-B4), Fragment Length, and Epoch (§III-A3) in TinyDTLS. To validate whether symbolic execution unguided by assumptions as used in SYMBEXNET could indeed miss such bugs, we symbolically executed MbedTLS and TinyDTLS without any assumptions or assertions. In all cases, none of the generated test inputs exposed the corresponding requirement violation.

Pedrosa *et al.* [24] use symbolic execution to search for interoperability issues by characterizing messages that during a session can be sent by one party but rejected as non-compliant by the other. Wen *et al.* [37] apply symbolic execution to stateful protocol implementations by first employing model learning (L^*) to extract a finite state machine, whose states can be considered as an additional input in symbolic execution.

Chau *et al.* [9], [10] use symbolic execution for a form of differential testing of libraries that classify certificate chains as valid or invalid. Using symbolic execution, path constraints for the “valid” and “invalid” outcomes are generated. Path constraints from different implementations are compared, and discrepancies are investigated. This approach is suitable for libraries that implement an input-output function for which requirements are difficult to formalize precisely. In our approach, we can formalize and check each requirement individually: violations can be detected and diagnosed directly when they occur, and there is no need to compare potentially complex test data from different implementations.

Chen *et al.* [11] automate the extraction of requirements from the X.509 RFC [4] by extracting sentences with keywords, such as “MUST”, “SHOULD”, etc. From each requirement, “valid” and “invalid” field values are constructed, with manual assistance. These are combined into a compact test suite, which is then used in differential black-box testing. As we have noted in §II, in the case of DTLS, some requirements are clarified and/or corrected in errata, which are documents distinct from the protocol’s RFC. It is unclear how requirement extraction can be automated in such a setting.

There is a rich body of work on protocol conformance testing, with model-based testing (MBT) [6], [35] and property-based testing (PBT) [1], [20], [21] as closely related approaches. An abstract model of the protocol is manually constructed or learned [33], and is used as basis for generating test inputs that are supplied, typically in black-box testing. Many different formalisms for expressing abstract models and specifications have been suggested [22], [36]. PBT is often simpler to perform than MBT, due to the high-level infrastructure and built-in mechanisms for input generation that PBT tools provide. Our

approach is white-box, which gives more power to the search for specific inputs and code paths that expose bugs.

State fuzzing is a black-box technique for detecting flaws in the control logic for handling the order of packets. Such flaws may be exploited, e.g., by tricking an implementation to bypass an authentication step. State fuzzing has discovered several flaws in TLS and DTLS implementations [3], [13], [16], [31]. It can be regarded as complementary to our approach in that it tests different orderings of packet types, whereas we search for requirement violations and bugs in handling packet fields under a specific ordering.

VII. REFLECTIONS AND CONCLUDING REMARKS

In this paper, we have described a test methodology and presented our experiences from using symbolic execution to detect specification violations and security vulnerabilities in implementations of the DTLS network protocol. A central idea in our methodology is to formulate requirements over the processing of packets in a session and let these requirements guide the symbolic execution to search for code paths and corresponding sequences of inputs that expose requirement violations. This allows testing the SUT with a broad range of input sequences, both benign and adversarial.

When testing a code base of significant size and complexity, a challenge for all techniques based on symbolic execution, and to those using KLEE in particular, is scalability. Our experience is that one effective way to achieve scalability is to test for violation of only one requirement at a time, and to make symbolic only the parts of the input that are relevant for the tested requirement. Naturally, this requires attentiveness, and has the drawback that it can miss inputs that can be produced only when multiple requirements are in effect.

If the speed of symbolic execution is slow due to some easily identifiable reason (e.g., the execution of cryptographic functions), one can try to find a way to keep away from such trouble when the requirement to check does not depend on the functionality of the corresponding code. This is what we described in §IV-B: we focused on handshakes with a pre-shared key (PSK) instead of testing the DTLS implementations using handshakes based on more complicated encryption algorithms. Of course, a downside is that testing will not detect bugs in the code which is not executed; it will only check violations of specification requirements.

When testing several implementations of the same protocol, significant effort can be saved by packaging as much reusable functionality as possible into a shared library, as we describe in §IV-C. By making the different implementations take input with a common structure, as we did for DTLS records, many operations on input data, such as making parts of input symbolic and adding assumptions corresponding to requirements, can be implemented in such a shared library.

We have evaluated our methodology by checking four widely-used DTLS implementations against the RFC for DTLS. We were able to quickly expose a known CVE in an older version of OpenSSL, and discover numerous new vulnerabilities, non-conformance issues and subtle errors in all implementations.

REFERENCES

- [1] T. Arts, J. Hughes, J. Johansson, and U. Wiger, “Testing telecoms software with Quviq QuickCheck,” in *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*. New York, NY, USA: ACM, 2006, pp. 2–10. [Online]. Available: <http://doi.acm.org/10.1145/1159789.1159792>
- [2] H. Asadian, P. Fiterău-Broștean, B. Jonsson, and K. Sagonas, “Replication material for the ICST 2022 paper: Applying symbolic execution to test implementations of a network protocol against its specification,” Apr. 2022. [Online]. Available: <https://doi.org/10.5281/zenodo.5929867>
- [3] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue, “A messy state of the union: Taming the composite state machines of TLS,” *Commun. ACM*, vol. 60, no. 2, pp. 99–107, Feb. 2017. [Online]. Available: <https://doi.org/10.1145/3023357>
- [4] S. Boeyen, S. Santesson, T. Polk, R. Housley, S. Farrell, and D. Cooper, “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile,” RFC 5280, May 2008. [Online]. Available: <https://rfc-editor.org/rfc/rfc5280.txt>
- [5] S. Bradner, “Key words for use in RFCs to Indicate Requirement Levels,” RFC 2119, Mar. 1997. [Online]. Available: <https://www.rfc-editor.org/info/rfc2119>
- [6] M. Broy, B. Jonsson, J. Katoen, M. Leucker, and A. Pretschner, Eds., *Model-Based Testing of Reactive Systems, Advanced Lectures*, ser. LNCS, vol. 3472. Springer, 2005. [Online]. Available: <https://doi.org/10.1007/b137241>
- [7] C. Cadar, D. Dunbar, and D. Engler, “KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI ’08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1855741.1855756>
- [8] M. M. Carvalho, J. DeMott, R. Ford, and D. A. Wheeler, “Heartbleed 101,” *IEEE Secur. Priv.*, vol. 12, no. 4, pp. 63–67, 2014. [Online]. Available: <https://doi.org/10.1109/MSP.2014.66>
- [9] S. Y. Chau, O. Chowdhury, M. E. Hoque, H. Ge, A. Kate, C. Nita-Rotaru, and N. Li, “SymCerts: Practical symbolic execution for exposing noncompliance in X.509 certificate validation implementations,” in *2017 IEEE Symposium on Security and Privacy*, ser. SP 2017. IEEE Computer Society, May 2017, pp. 503–520. [Online]. Available: <https://doi.org/10.1109/SP.2017.40>
- [10] S. Y. Chau, M. Yahyazadeh, O. Chowdhury, A. Kate, and N. Li, “Analyzing semantic correctness with symbolic execution: A case study on PKCS#1 v1.5 signature verification,” in *26th Annual Network and Distributed System Security Symposium*, ser. NDSS 2019. The Internet Society, Feb. 2019. [Online]. Available: <https://www.ndss-symposium.org/ndss-paper/analyzing-semantic-correctness-with-symbolic-execution-a-case-study-on-pkcs1-v1-5-signature-verification/>
- [11] C. Chen, C. Tian, Z. Duan, and L. Zhao, “RFC-directed differential testing of certificate validation in SSL/TLS implementations,” in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE 2018, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, May–Jun. 2018, pp. 859–870. [Online]. Available: <https://doi.org/10.1145/3180155.3180226>
- [12] V. Chipounov, V. Kuznetsov, and G. Candea, “The S2E platform: Design, implementation, and applications,” *ACM Trans. Comput. Syst.*, vol. 30, no. 1, pp. 2:1–2:49, 2012. [Online]. Available: <https://doi.org/10.1145/2110356.2110358>
- [13] J. de Ruiter and E. Poll, “Protocol state fuzzing of TLS implementations,” in *24th USENIX Security Symposium*. USENIX Association, Aug. 2015, pp. 193–206. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter>
- [14] T. Dierks and E. Rescorla, “The transport layer security (TLS) protocol version 1.1,” RFC 4346, Internet Engineering Task Force, Apr. 2006. [Online]. Available: <http://www.ietf.org/rfc/rfc4346.txt>
- [15] —, “The transport layer security TLS protocol version 1.2,” RFC 5246, Aug. 2008. [Online]. Available: <http://www.rfc-editor.org/rfc/rfc5246.txt>
- [16] P. Fiterău-Broștean, B. Jonsson, R. Merget, J. de Ruiter, K. Sagonas, and J. Somorovsky, “Analysis of DTLS implementations using protocol state fuzzing,” in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2523–2540. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean>
- [17] P. Fiterău-Broștean, B. Jonsson, K. Sagonas, and F. Tåquist, “DTLS-Fuzzer: A DTLS protocol state fuzzer,” in *15th IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST 2022. IEEE Computer Society, Apr. 2022.
- [18] F. Foerg, “Fuzzing the Mbed TLS library,” Sep. 2015. [Online]. Available: <https://blog.gdssecurity.com/labs/2015/9/21/fuzzing-the-mbed-tls-library.html>
- [19] J. C. King, “Symbolic execution and program testing,” *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976. [Online]. Available: <https://doi.org/10.1145/360248.360252>
- [20] A. Löscher and K. Sagonas, “Targeted property-based testing,” in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 46–56. [Online]. Available: <http://doi.acm.org/10.1145/3092703.3092711>
- [21] A. Löscher, K. Sagonas, and T. Voigt, “Property-based testing of sensor networks,” in *Sensing, Communication, and Networking, 12th Annual IEEE International Conference on*, ser. SECON ’15. IEEE, Jun. 2015, pp. 100–108. [Online]. Available: <https://doi.org/10.1109/SAHCN.2015.7338296>
- [22] K. L. McMillan and L. D. Zuck, “Formal specification and testing of QUIC,” in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM 2019. ACM, Sep. 2019, pp. 227–240. [Online]. Available: <https://doi.org/10.1145/3341302.3342087>
- [23] B. Möller, T. Duong, and K. Kotowicz, “This POODLE bites: exploiting the SSL 3.0 fallback,” 2014. [Online]. Available: <https://www.openssl.org/~bodo/ssl-poodle.pdf>
- [24] L. Pedrosa, A. Fogel, N. Kothari, R. Govindan, R. Mahajan, and T. D. Millstein, “Analyzing protocol implementations for interoperability,” in *12th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI 15. USENIX Association, May 2015, pp. 485–498. [Online]. Available: <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pedrosa>
- [25] V.-T. Pham, M. Böhme, and A. Roychoudhury, “AFLNET: A greybox fuzzer for network protocols,” in *IEEE 13th International Conference on Software Testing, Validation and Verification*, ser. ICST 2020. IEEE, Oct. 2020, pp. 460–465. [Online]. Available: <https://ieeexplore.ieee.org/document/9159093>
- [26] E. Rescorla and N. Modadugu, “Datagram transport layer security,” RFC 4347, Internet Engineering Task Force, Apr. 2006. [Online]. Available: <http://www.ietf.org/rfc/rfc4347.txt>
- [27] —, “Datagram transport layer security version 1.2,” RFC 6347, pp. 1–32, Jan. 2012. [Online]. Available: <http://www.rfc-editor.org/rfc/rfc6347.txt>
- [28] R. Sasnauskas, P. Kaiser, R. L. Jukic, and K. Wehrle, “Integration testing of protocol implementations using symbolic distributed execution,” in *20th IEEE International Conference on Network Protocols*, ser. ICNP 2012. IEEE Computer Society, Oct.–Nov. 2012, pp. 1–6. [Online]. Available: <https://doi.org/10.1109/ICNP.2012.6459940>
- [29] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle, “KleeNet: discovering insidious interaction bugs in wireless sensor networks before deployment,” in *Proceedings of the 9th International Conference on Information Processing in Sensor Networks*, ser. IPSN 2010, T. F. Abdelzaher, T. Voigt, and A. Wolisz, Eds. ACM, Apr. 2010, pp. 186–196. [Online]. Available: <https://doi.org/10.1145/1791212.1791235>
- [30] Z. Shelby, K. Hartke, and C. Bormann, “The constrained application protocol (CoAP),” RFC 7252, Jun. 2014. [Online]. Available: <https://rfc-editor.org/rfc/rfc7252.txt>
- [31] J. Somorovsky, “Systematic fuzzing and testing of TLS libraries,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’16. New York, NY, USA: ACM, 2016, pp. 1492–1504. [Online]. Available: <http://doi.acm.org/10.1145/2976749.2978411>
- [32] J. Song, C. Cadar, and P. R. Pietzuch, “SYMBEXNET: Testing network protocol implementations with symbolic execution and rule-based specifications,” *IEEE Trans. Software Eng.*, vol. 40, no. 7, pp. 695–709, 2014. [Online]. Available: <https://doi.org/10.1109/TSE.2014.2323977>
- [33] M. Tappler, B. K. Aichernig, and R. Bloem, “Model-based testing IoT communication via active automata learning,” in *Software Testing, Verification and Validation, (ICST) 2017 IEEE International Conference on*. IEEE Computer Society, Mar. 2017, pp. 276–287. [Online]. Available: <https://doi.org/10.1109/ICST.2017.32>

- [34] "TLS-Attacker," <https://github.com/tls-attacker/TLS-Attacker>, Online; accessed 12-December-2021.
- [35] M. Utting and B. Legeard, *Practical Model-Based Testing - A Tools Approach*, 1st ed. Morgan Kaufmann, Nov. 2006. [Online]. Available: <http://www.elsevierdirect.com/product.jsp?isbn=9780123725011>
- [36] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, "Model-based testing of object-oriented reactive systems with Spec Explorer," in *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, ser. LNCS, R. M. Hierons, J. P. Bowen, and M. Harman, Eds., vol. 4949. Springer, 2008, pp. 39–76. [Online]. Available: https://doi.org/10.1007/978-3-540-78917-8_2
- [37] S. Wen, Q. Meng, C. Feng, and C. Tang, "A model-guided symbolic execution approach for network protocol implementations and vulnerability detection," *PloS one*, vol. 12, no. 11, p. e0188229, 2017. [Online]. Available: <https://doi.org/10.1371/journal.pone.0188229>
- [38] M. Yason, "CVE-2014-0195: Adventures in OpenSSL's DTLS fragmented land," Dec. 2014. [Online]. Available: <https://securityintelligence.com/cve-2014-0195-adventures-in-openssl-dtls-fragmented-land/>
- [39] M. Zalewski, "American fuzzy lop," <http://lcamtuf.coredump.cx/afl/>, 2013.