DTLS-Fuzzer: A DTLS Protocol State Fuzzer

Paul Fiterău-Broștean, Bengt Jonsson, Konstantinos Sagonas, Fredrik Tåquist
Uppsala University, Uppsala, Sweden
{paul.fiterau_brostean, bengt, kostis, fredrik.takvist}@it.uu.se

Abstract—DTLS-Fuzzer is a protocol state fuzzer for implementations of DTLS clients and servers. DTLS-Fuzzer uses model learning to generate a state machine model of a DTLS implementation, capturing its input/output behavior. This model can be used for model-based testing or can be analyzed for security vulnerabilities and specification violations. This demo abstract overviews the architecture, API, and usage of the tool.

Index Terms—model learning, network security testing, model-based testing

I. INTRODUCTION

Protocol state fuzzing is a testing technique which has been used successfully to test implementations of critical protocols such as TLS [4], SSH [9], DTLS [7] and QUIC [5]. The technique uses model learning [18] to automatically infer state machines of protocol implementations. Model learning is an automated black-box technique in which selected sequences of messages are sent to the implementation, in order to produce a Mealy machine capturing how the implementation reacts to message flows. This Mealy machine is then analyzed to spot flaws in the implementation’s control logic. Protocol state fuzzing requires a protocol-specific test harness, aka a Mapper, which translates between symbols in the Mealy machine and packets exchanged with the System Under Test (SUT).

DTLS-Fuzzer [2] implements a protocol state fuzzing framework for DTLS [15] and, in doing so, tackles the challenges that developing such a framework entails. Constructing a Mapper is often difficult and time-consuming, particularly for complex protocols such as DTLS. DTLS-Fuzzer provides a Mapper that works for both DTLS clients and servers. Built on top of TLS-Attacker [16], [17], an extensible (D)TLS test framework, the Mapper supports different key exchange algorithms and components, which are shown in Fig. 1. The Learner, the component that now includes support for DTLS clients.

II. ARCHITECTURE

At a high level, DTLS-Fuzzer comprises two main components, which are shown in Fig. 1. The Learner, the component implementing model learning, is based on LearnLib [3], [12], a Java library implementing various model learning algorithms.

DTLS-Fuzzer implements input symbols using the class hierarchy shown in Fig. 2. The abstract class TlsInput provides an abstract representation for all input symbols. There are two kinds of TlsInput subclasses implementing input symbols. Specific subclasses, such as ServerHelloInput, can be used to define input symbols for specific DTLS messages. The handling of these symbols has been specifically adjusted for model learning (e.g., so that more compact models are inferred). The GenericTlsInput subclass, on the other hand, can be used to define symbols for a wide range of DTLS messages. DTLS-Fuzzer applies TLS-Attacker’s handling for sending these messages, which lacks model learning adjustments but is often sufficient to generate usable models.

With respect to output processing, the Mapper forms output symbols by extracting the types of the message received from the SUT. Strings describing these types (e.g., “ServerHello”) are then encapsulated in a TlsOutput instance and provided to the Learner. Exceptional cases arise when no message is received or the message cannot be decrypted. For each such case the Mapper produces a specific TlsOutput.
III. INTERFACE

DTLS-Fuzzer requires an *input alphabet*, which specifies the input symbols to use during model learning. The alphabet is provided in an XML file. When processed, the file is unmarshalled using the JAXB framework [14], resulting in a collection of TlsInput instances, one for each child element of the root element alphabet.

```xml
<alphabet>
    <HelloVerifyRequestInput name="HelloVerifyRequest"/>
    <ServerHelloInput suite="TLS.PSK.WITH.AES.128.CCM"
        name="PskServerHello"/>
    <ServerHelloDoneInput name="ServerHelloDone"/>
    <ChangeCipherSpecInput name="ChangeCipherSpec"/>
    <FinishedInput name="Finished"/>
    <GenericTlsInput name="Application"/>
    <GenericTlsInput name="Alert"/>
</alphabet>
```

Displayed above is a typical alphabet we would use to test DTLS clients. It defines input symbols for seven distinct messages (e.g., HelloVerifyRequest). The names of alphabet’s child elements correspond to those of the classes that implement the symbols, the attributes correspond to fields of these classes. The common attribute, name, determines the label under which the symbol will appear in the generated state machine model. GenericTlsInput elements contain a specification of the encapsulated DTLS message which is highly customizable; most message fields can be adjusted (e.g., by setting them to fixed values), otherwise DTLS-Fuzzer will automatically choose appropriate values.

*Learning Parameters:* A user can select algorithms for hypothesis generation (e.g., TTT [11]) and hypothesis validation (e.g., Random Walk [13]), the two core steps of model learning. For each algorithm, the user can configure associated parameters such as the bound on the number of tests executed before a hypothesis is deemed “correct”, and returned as the learned model.

*SUT Parameters:* The main SUT parameters are: 1) the command used to execute the SUT (which differs between implementations); 2) the time to wait for the SUT to start before sending messages to it; 3) the time to wait for each response. The last two parameters are important in ensuring that the output presented to the time-agnostic Learner depends uniquely on the input sequence and is not affected by time effects. Such could be the case if, e.g., after sending a message, we do not give the SUT sufficient time to produce a response.

IV. USAGE

In the typical use-case scenario, DTLS-Fuzzer is applied on a specific SUT. This involves the following steps: 1) building DTLS-Fuzzer; 2) deploying the SUT; 3) running DTLS-Fuzzer with appropriate arguments; 4) (manually) analyzing models. Because of DTLS-Fuzzer’s support infrastructure, these steps can often be performed using four commands from DTLS-Fuzzer’s parent directory. As an example, we show commands that perform these steps to test MbedTLS [1], a (D)TLS library commonly used to secure embedded applications.

We start by building DTLS-Fuzzer using the prepare.sh script (Line 1), which downloads TLS-Attacker 3.0b (the version DTLS-Fuzzer relies on), and uses mvn to install TLS-Attacker and DTLS-Fuzzer. We then use the setup_sut.sh script to deploy MbedTLS 2.26.0 (Line 2). The script automates deployment for nine distinct libraries. Next, we launch DTLS-Fuzzer with arguments for its parameters. Arguments can be provided by command-line or, as done on Line 4, through an argument file. DTLS-Fuzzer contains over 50 pre-made argument files for common learning experiments. As their names suggest, these files are parameterized by the SUT they are designed for, the alphabet configuration (e.g., the key exchange algorithm the input alphabet uses) and the algorithm used for hypothesis validation. Other arguments, such as the algorithm used for hypothesis construction, almost never have to be adjusted. Once DTLS-Fuzzer finishes learning, we can visualize the model using e.g., xdot [10] (Line 5). The experiment ends with manual analysis of the model for bugs. Our demo video [6] revisits this scenario, and shows how supporting scripts can ease model analysis. Similar steps are performed to test other SUTs, noting that the input alphabet can be manually constructed or selected from DTLS-Fuzzer’s pre-made alphabets, and should include messages that both DTLS-Fuzzer and the SUT support. Also, to execute the SUT we need DTLS client/server programs. Often, such programs come packaged with the SUT.

V. CHALLENGES AND FUTURE WORK

We end this brief demo abstract with some challenges and future development plans. Currently, time parameters have to be set carefully to avoid unwanted time effects which may hinder model learning. In the worst case, one may even have to make changes to the SUT to make it less time-sensitive. DTLS-Fuzzer provides several mitigations, e.g., discarding duplicate messages or executing an input sequence multiple times, and only returning the most common output sequence. This makes learning more robust, but does not completely avoid the problem, which, at its core, lies in the use of a time-agnostic learning algorithm to infer a time-dependent SUT.

Another area for improvement stems the fact that the models DTLS-Fuzzer produces have to be analyzed manually, which is a time-consuming task, requires protocol knowledge, and can easily miss bugs. DTLS-Fuzzer provides scripts that aid in model visualization. Still, these cannot replace a mechanism for automatic bug detection. We have recently devised such a mechanism wherein state machine bugs are specified as finite automata [8], and then automatically identified on an automaton derived from the SUT’s model using automata intersection. We plan to extend DTLS-Fuzzer with this mechanism, to further enhance its support for testing automation.

```
1 $ bash prepare.sh
2 $ bash setup_sut.sh mbedtls-2.26.0
3 $ java -jar target/dtls-fuzzer.jar \
4    args/mbedtls/learn mbedtls.client.psk.rwalk
5 $ xdot output/mbedtls.../learnedModel.dot
```
REFERENCES