





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 certificate authentication settings. This support proved crucial in a recent application of DTLS-Fuzzer to test widely used DTLS server implementations [7]. The application produced models exposing bugs (incl. vulnerabilities) in all nine libraries tested, and resulted in fixes to five of them. This demo abstract and associated video [6] present a version of DTLS-Fuzzer, 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.

Work partially funded by the Swedish Research Council (Vetenskapsrådet), the Swedish Foundation for Strategic Research through project aSSIsT, and the Knut and Alice Wallenberg Foundation through project UPDATE.

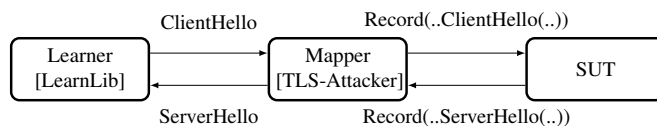


Fig. 1. Protocol state fuzzing setup implemented in DTLS-Fuzzer

The Mapper, implemented using TLS-Attacker, mediates the Learner’s communication with the SUT.

Upon receiving an input symbol from the Learner, the Mapper instantiates the corresponding message, encapsulates it in a DTLS record (the message carrier in DTLS), sends the record to the SUT, gathers the SUT’s response, extracts the DTLS messages, and returns the corresponding output symbols. In order to translate between symbols and DTLS records, DTLS-Fuzzer’s Mapper maintains the state of its interaction with the SUT, which we will refer to as the *execution context*. For each input symbol, the Mapper creates the corresponding DTLS message, updates and uses its state to configure message fields. The Mapper updates its state again when receiving a response from the SUT.

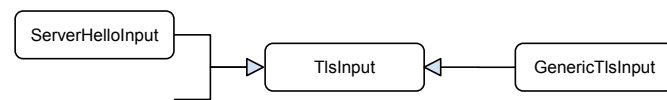


Fig. 2. Class hierarchy implementing input symbols

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`.

```
<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"> <Application/>
</GenericTlsInput>
  <GenericTlsInput name="Alert"> <Alert/>
</GenericTlsInput>
</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.

```
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
```

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.

REFERENCES

- [1] ARM, “arm MBED,” <https://tls.mbed.org/>, Online; accessed 12-December-2021.
- [2] aSSiST Project, “DTLS-Fuzzer,” <https://github.com/assist-project/dtls-fuzzer>, Online; accessed 17-December-2021.
- [3] Chair for Programming Systems, “LearnLib,” <https://learnlib.de/>, TU Dortmund University, Online; accessed 17-December-2021.
- [4] 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>
- [5] T. Ferreira, H. Brewton, L. D’Antoni, and A. Silva, “Prognosis: Closed-box analysis of network protocol implementations,” in *ACM SIGCOMM 2021 Conference*. ACM, Aug. 2021, pp. 762–774. [Online]. Available: <https://doi.org/10.1145/3452296.3472938>
- [6] P. Fiterău-Broștean, “DTLS-Fuzzer Demo Video,” <https://youtu.be/KtEpwYC-f9M>, 2022.
- [7] 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>
- [8] P. Fiterău-Broștean, B. Jonsson, K. Sagonas, and F. Tåquist, “Automata-based automated detection of state machine bugs in protocol implementations,” 2022, submitted.
- [9] P. Fiterău-Broștean, T. Lenaerts, J. de Ruiter, E. Poll, F. W. Vaandrager, and P. Verleg, “Model learning and model checking of SSH implementations,” in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, ser. SPIN 2017. ACM, 2017, pp. 142–151. [Online]. Available: <https://doi.org/10.1145/3092282.3092289>
- [10] J. Fonseca, “xdot 1.2,” <https://pypi.org/project/xdot/>, Online; accessed 12-December-2021.
- [11] M. Isberner, F. Howar, and B. Steffen, “The TTT algorithm: A redundancy-free approach to active automata learning,” in *Runtime Verification: 5th International Conference, RV 2014, Proceedings*, ser. LNCS. Springer, Sep. 2014, vol. 8734, pp. 307–322. [Online]. Available: https://doi.org/10.1007/978-3-319-11164-3_26
- [12] —, “The open-source LearnLib - A framework for active automata learning,” in *Computer Aided Verification - 27th International Conference, CAV, ser. LNCS*, vol. 9206. Springer, 2015, pp. 487–495. [Online]. Available: https://dx.doi.org/10.1007/978-3-319-21690-4_32
- [13] D. Lee and M. Yannakakis, “Principles and methods of testing finite state machines—a survey,” *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, 1996. [Online]. Available: <https://ieeexplore.ieee.org/document/533956>
- [14] Oracle, “JAXB. Java Architecture for XML Binding,” <https://javaee.github.io/jaxb-v2/>, Online; accessed 12-December-2021.
- [15] E. Rescorla and N. Modadugu, “Datagram transport layer security version 1.2,” RFC 6347, pp. 1–32, Jan. 2012. [Online]. Available: <http://www.rfc-editor.org/rfc/rfc6347.txt>
- [16] 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>
- [17] “TLS-Attacker,” <https://github.com/tls-attacker/TLS-Attacker>, Online; accessed 12-December-2021.
- [18] F. W. Vaandrager, “Model learning,” *Commun. ACM*, vol. 60, no. 2, pp. 86–95, 2017. [Online]. Available: <https://doi.org/10.1145/2967606>