EDHOC-Fuzzer: An EDHOC Protocol State Fuzzer

Konstantinos Sagonas
kostis@it.uu.se
Uppsala University
Uppsala, Sweden
National Technical University of Athens
Athens, Greece

Thanasis Typaldos
actypaldos@gmail.com
National Technical University of Athens
Athens, Greece

ABSTRACT

EDHOC is a compact and lightweight authenticated key exchange protocol proposed by the IETF, whose design focuses on small message sizes, in order to be suitable for constrained IoT communication technologies. In this tool paper, we overview EDHOC-Fuzzer, a protocol state fuzzer for implementations of EDHOC clients and servers. It employs model learning to generate a state machine model of an EDHOC implementation, capturing its input/output behavior. This model can then be used for model-based testing, for fingerprinting, or can be analyzed for non-conformances, state machine bugs and security vulnerabilities. We overview the architecture and use of EDHOC-Fuzzer, and present some examples of models produced by the tool and our current findings.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Security and privacy** → **Software security engineering**.

KEYWORDS

Software security, protocol security, model learning, model-based testing, fuzzing, IoT protocols, EDHOC, OSCORE, CoAP

ACM Reference Format:
Konstantinos Sagonas and Thanasis Typaldos. 2023. EDHOC-Fuzzer: An EDHOC Protocol State Fuzzer. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3597926.3604922

1 INTRODUCTION

For many computer applications, the reliability and security of communication and network protocols is of utmost importance but can easily be compromised by flaws in their implementations. A crucial aspect of a protocol's implementation is the correct and robust design of its underlying state machine. Any deviation from the order prescribed in the protocol's specification may constitute anything between an inconsequential error to a serious vulnerability. Corresponding implementation flaws, called *state machine bugs*, may be exploitable, e.g., to bypass authentication steps, establish insecure connections, or reveal sensitive data to a malicious attacker.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3604922

Using black-box testing techniques from formal methods, we can automatically infer a quite accurate model of an implementation's state machine. Such techniques are known under a plethora of different names: *regular inference* [2], *active automata learning* [20], *model learning* [21] or, in the case of protocol implementations, *protocol state fuzzing* [5, 9]. (We will stick to the last two terms and use them interchangeably in what follows.) This inferred state machine provides useful insights into the choices and errors made in the protocol's implementation, and can be analyzed for detection of logical flaws and for compliance of the implementation with the protocol's specification. Its ocular inspection can also reveal superfluous states or extraneous transitions that the developers never intended to include and, consequently, should be removed from the implementation as a precaution. Furthermore, this inferred state machine gives a good overview of the sequences of messages exchanged, and can be used to automatically detect state machine bugs in a protocol's implementation [8].

In this tool demostration paper, we present EDHOC-Fuzzer, a protocol state fuzzer for EDHOC implementations. EDHOC [16] is a compact and lightweight key exchange protocol recently proposed by the Intenet Engineering Task Force (IETF). Its main target is that of low-power communication technologies, such as those used in the IoT domain. Like similar tools for other network security protocols (e.g., TLS-Attacker [19], DTLS-Fuzzer [7], and Prognosis [6]), EDHOC-Fuzzer uses model learning to generate a Mealy machine model of an EDHOC client or server implementation capturing its input/output behavior.

After presenting the necessary background (§2), we overview the tool's architecture and the typical steps of using EDHOC-Fuzzer in §3, and present some findings from applying the tool to learn state machine models of three existing EDHOC implementations in §4. The paper ends with a brief section of related work (§5) and some concluding remarks.

2 BACKGROUND

2.1 OSCORE and EDHOC

OSCORE [17] (Object Security for Constrained RESTful Environments), defines a protection method for CoAP [18] (Constrained Application Protocol), using CBOR [3] (Concise Binary Object Representation) and COSE [15] (CBOR Object Signing and Encryption). OSCORE is designed to provide end-to-end security between two CoAP endpoints and requires from the participants to derive and establish a security context. In order for this to happen, the necessary information and keying material should be exchanged in a secure and authenticated way, provided by a suitable key exchange protocol such as EDHOC.

EDHOC [16], the Ephemeral Diffie-Hellman Over COSE protocol, is a compact and lightweight authenticated key exchange protocol. One of its main use cases is to provide secure key exchange for OSCORE. In fact, EDHOC uses the same primitives as OSCORE: COSE for cryptography, CBOR for encoding, and CoAP for transport. Being a stateful protocol, its implementations need to maintain a state machine which stores information about how far a protocol session between peers has progressed.

In EDHOC, peers can have two roles: *Initiator* or *Responder*. These roles are not tied to the web transfer protocol used. For instance, when CoAP is used, a CoAP client can either be an Initiator or a Responder for the EDHOC protocol.

The EDHOC protocol consists of five messages (M1..M4, ERR_E). The Initiator uses M1 and M3, while the Responder uses M2 and, optionally, M4. Both roles can use ERR_E, which signals an error.

2.2 Model Learning

Model learning [21] is an automated black-box technique which needs to know the input and output alphabets of the System Under Test (SUT). In addition, some abstraction is often needed, i.e., a way to map concrete protocol messages to abstract alphabet symbols, and vice versa, in order to interact with the SUT. This functionality is provided by a protocol-specific component called the *mapper*. Most model learning algorithms (e.g., the classic L^* algorithm [2] or those provided by LearnLib [12], the library which we use) produce a deterministic Mealy machine as a model by operating in two alternating phases: *hypothesis construction* and *hypothesis validation*.

During hypothesis construction, learning sends sequences of input symbols (*Is*) to the SUT, observing the sequences of output symbols (*Os*) that are generated in response. When certain convergence criteria are met, learning constructs a *hypothesis* H, which is a minimal deterministic Mealy machine that is consistent with the *Os* for the *Is* that have been sent to the SUT. For the remaining inputs, H predicts corresponding outputs by extrapolating from the observed *Os*. To validate that these predictions agree with the behavior of the SUT, learning then moves to a validation phase in which the SUT is subject to a conformance testing algorithm which aims to validate that the SUT's behavior agrees with H. If conformance testing does not find any counterexample, learning returns the current hypothesis as the inferred model M of the SUT. If a counterexample (i.e., an I on which the SUT and H disagree) is found, the hypothesis construction phase is re-entered to build a more refined hypothesis H' which also takes that I into account. If the loop of hypothesis construction and validation does not terminate, this indicates that the behavior of the SUT cannot be captured by a deterministic Mealy machine whose size and complexity is within reach of the employed learning algorithm. Still, even in these cases, the last constructed hypothesis can be used as an *approximate* model of the SUT's state machine.

3 EDHOC-FUZZER

3.1 Architecture

An overview of the EDHOC-Fuzzer's architecture is shown in fig. 1, which depicts the System Under Test as a black box because it is treated as such. The component that provides the core of EDHOC-Fuzzer's functionality is ProtocolState-Fuzzer, which is a generic

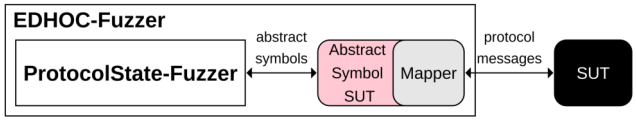

Figure 1: The EDHOC-Fuzzer consists of the ProtocolState-Fuzzer connected with the Abstract Symbol SUT, which uses the Mapper to communicate through the network with the black-box SUT.

and protocol agnostic framework for protocol state fuzzing that we have also developed and will release as open source. Its implementation uses the open-source LearnLib library [12] for active automata learning. EDHOC-Fuzzer's second main component is the Mapper, which is protocol specific. Its purpose is to i) convert an abstract input symbol to a concrete protocol message and send it to the SUT, and ii) receive the SUT's concrete protocol message and convert it to an abstract output symbol. The Mapper is capable of handling the protocol's state and reading and writing CoAP, EDHOC and OSCORE messages using the `cf-edhoc` module in the edhoc branch of the Eclipse Californium implementation of CoAP for Java after some modifications. The third and last main part of EDHOC-Fuzzer is the Abstract Symbol SUT, a glue component which connects the other two.

3.2 Ease of Use

To show the ease of use of the tool, we present the typical use of EDHOC-Fuzzer which consists of the following steps: 1) set up EDHOC-Fuzzer; 2) set up the SUT of interest; 3) run the EDHOC-Fuzzer with appropriate arguments for the SUT; and 4) optionally, make the learned model more readable. We provide an example that showcases these steps to learn the model of Californium's EDHOC server implementation (which we will abbreviate as RISE).

```
$ scripts/setup_fuzzer.sh -l
$ scripts/setup_sul.sh rise
$ java -jar edhoc-fuzzer.jar experiments/args/rise/server
$ scripts/beautify_model.sh experiments/results/servers/rise/learnedModel.dot
```

Users can employ the `setup_fuzzer.sh` script to build EDHOC-Fuzzer. Subsequently, the `setup_sul.sh` script can be used to download and setup the RISE SUT, using a specific point in its commit history. Alternatively, users can modify this script with another commit point or add the SUT of their interest by specifying the location to download it from, its dependencies, and the commands to build it. The next step is to run the EDHOC-Fuzzer by supplying appropriate arguments for the SUT. EDHOC-Fuzzer comes with pre-packaged argument files for several EDHOC implementations, including the RISE server. These arguments can control various aspects of the SUT's usage (e.g., the port to use, the time to wait for a response, etc.), of the protocol support (e.g., which RFC version the SUT implements), and of the model learning (e.g., the learning and the equivalence algorithm, their parameters, etc.). After model learning has successfully terminated, the learned model (in both `DOT` and `PDF` format) is stored in a subdirectory of `experiments/results` directory as specified in the arguments file. If desired, this model can be made more readable using a provided script that merges same transitions and shortens symbol names.

4 USE CASES

In this section, we present three examples of using the tool to learn models of SUTs and some of the issues that model inspection allowed us to identify. The implementations we used (uOSCORE-uEDHOC, RISE and EDHOC-RS) are all open source, follow the changes in the protocol, and are actively maintained. Interestingly, they are implemented in three different languages: C, Java and Rust, respectively.

The shown models are those produced by the `beautify_model.sh` script but we also use colors for better illustration of good and erroneous paths. Their transitions, which are the result of merging several transitions in the original learned model, are labeled with sequences of elements of the form I/O, meaning that the EDHOC-Fuzzer sent the input symbol I and the SUT replied with symbol O. In the figures, we show these sequences with their elements one below the other. For example, in fig. 2, the learned model had seven edges from state 0 to state 1, each with one label, but these have been merged by the script to one edge with seven labels. Besides the five EDHOC messages mentioned in §2, we also use two special output symbols: ∅ to indicate that nothing was received, and ⊥ to indicate that the SUT has stopped listening to its specified port. In addition, APP_O is an OSCORE-protected application message created after the derivation of an OSCORE context, APP_C is the plain text version of APP_O (so it is not OSCORE-protected), EMP_C is a CoAP message with no payload, and M3APP_O is the concatenation of M3 and APP_O.

4.1 uOSCORE-uEDHOC Client

The learned model of uOSCORE-uEDHOC client (commit fbaa96c) is shown in fig. 2. The EDHOC-Fuzzer, which acts as a server, waits for the initial M1 from the client. It then sends M2 to the client, and receives back M3. As we can see, the state machine model transitions from state 0 to state 1. This concludes the exchange for both peers. The interesting part is that the client does not wait for a response to M3, but terminates immediately. In this case, according to the specification, the client should wait for an OSCORE message response, but this implementation does not handle OSCORE messages. We have reported this behavior to the developers, who acknowledged it (issue 48).

Figure 2: The learned model of uOSCORE-uEDHOC client.

4.2 RISE Client

The learned model of RISE client (commit f994359) is shown in fig. 3. Here we test the SUT configuration in which M4 is required and OSCORE messages are also exchanged.

In the 'normal' EDHOC exchange (path 0 → 2 → 4 → 1), the EDHOC-Fuzzer, which acts as a server, waits for the initial M1 from the client, then sends M2 and receives back M3 (transition 0 → 2). It then sends M4, which finishes the EDHOC exchange. The client is also configured to send an OSCORE message, so it prepares it after having derived the necessary OSCORE context and replies with APP_O (transition 2 → 4). In state 4, the client waits for a reply to its APP_O message (without replying back, thus the client-side symbol is ∅ in all transitions 4 → 1), and then terminates in state 1.

However, notice that there is an alternative EDHOC exchange (path 0 → 2 → 3 → 4 → 1). The EDHOC-Fuzzer received M1 from the client, sent M2 and received back M3. The model is at state 2, in which the client can derive an OSCORE context, but also expects M4 in order to successfully complete the EDHOC exchange. If it receives either M2 or M3 then it responds, correctly, with ERR_E (transition 2 → 3), but it does not terminate the exchange session as it should. Instead, the client having derived a new OSCORE context, replies with APP_O to any message received (transitions 3 → 4). This is a vulnerability, because it may expose the OSCORE context, which can be derived by both peers after receiving M3. We have privately reported this issue to the developers, who acknowledged and fixed it (commit fad09f2) making the client terminate the exchange session after sending ERR_E in state 3.

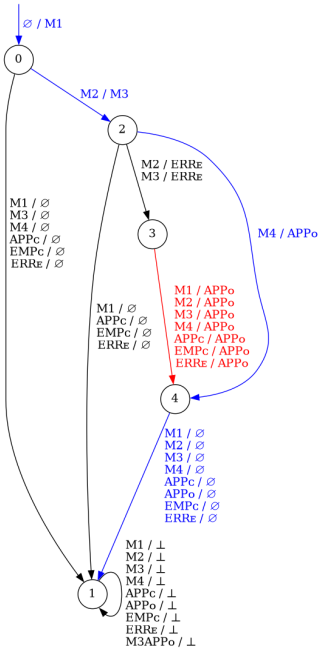

Figure 3: The learned model of RISE client.

4.3 EDHOC-RS Server

The learned model of EDHOC-RS server (commit 09aa2a8) is shown in fig. 4.

In the 'normal' EDHOC exchange (path 0 → 2 → 2), the EDHOC-Fuzzer sends M1 and receives M2 (transition 0 → 2) and then sends M3 and receives EMP_C (loop on state 2) terminating successfully the exchange. In this state, if the EDHOC-Fuzzer sends M1 then receives back M2 from the server starting a new EDHOC exchange (loop on state 2).

The learned model of fig. 4 also reveals two other interesting behaviors of the EDHOC-RS server. The first one is when the EDHOC-Fuzzer sends M3 and receives EMP_C (loop on state 2). This shows that after a successful EDHOC exchange, if the server receives another M3 it will reply with EMP_C instead of an error. This behavior is

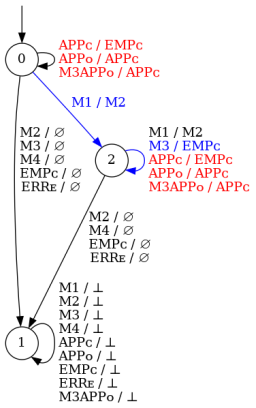

Figure 4: The learned model of EDHOC-RS server.

non-conforming to the draft protocol standard [16]. We have reported this to the developers (issue 76), who acknowledged and fixed the problem (commit 20a3e94).

The second non-conforming behavior is found in states 0 and 2. In these states, the EDHOC-RS server responds to APP_O, APP_C and M3APP_O with EMP_C and APP_C messages. These three input messages are sent to a CoAP resource that the server does not support. Thus, a standards-complying implementation of an EDHOC server should ignore such messages or reply with an error. We have also reported this to the developers (issue 77), who acknowledged and fixed it (commit f6b3b38) making the server reply with CoAP Error Messages (ERR_C).

5 RELATED WORK

Manually constructing models of network protocol implementations is tedious and error-prone. A better approach is to automatically construct an implementation's state machine either using model learning [21], as we do, or during fuzz testing.

Model learning techniques which have been applied to network protocols can be partitioned into approaches based on *passive learning* (e.g., [10, 11]), which learn an approximate protocol state machine from a set of network traces, and *active learning* ones (e.g., [4, 5, 9, 19]) that leverage some automata learning algorithm to construct a state machine by querying the implementation with appropriately constructed message sequences. A recent paper [1] presents a comparison between these two approaches.

As tools similar to EDHOC-Fuzzer, we mention TLS-Attacker [19] for TLS, DTLS-Fuzzer [7] for DTLS, and the Prognosis tool [6] which comes with support for the QUIC and TCP protocols.

In recent years, fuzz testing has been successful in detecting numerous crashes and vulnerabilities in non-stateful programs. Greybox fuzzers based on AFL that are aimed at testing network protocol implementations, such as AFLNET [14] or StateAFL [13], typically come with heuristics to build an approximation of the SUT's state machine, and use it to start the fuzzing from some (approximate) state and guide the generation of input sequences.

6 CONCLUDING REMARKS

In this tool demonstration paper, we overviewed EDHOC-Fuzzer, a protocol state fuzzer for implementations of the EDHOC authenticated key exchange protocol. EDHOC-Fuzzer is fully automatic, treats the provided EDHOC implementation as a black box, and employs model learning to generate a state machine model of the SUT capturing its input/output behavior. This model can provide useful insights into the choices and errors made in the protocol's implementation and it can be analyzed for non-conformance to the EDHOC protocol specification and for security vulnerabilities (as in the three use cases we presented). In a security evaluation, the inferred state machines can be hardened by removing superfluous states and transitions from them, thus simplifying the implementation and reducing the risk of vulnerabilities. Alternatively, the learned models can be used for model-based testing, for fingerprinting, or for automatic detection of state machine bugs by implementors and security researchers. Using EDHOC-Fuzzer, we intend to pursue some of these avenues for future research.

ACKNOWLEDGMENTS

This research has been supported in part by the Swedish Foundation for Strategic Research through the aSSIsT project.

REFERENCES

[1] Bernhard K. Aichernig, Edi Muskardin, and Andrea Pferscher. 2022. Active vs. Passive: A Comparison of Automata Learning Paradigms for Network Protocols. In *FMAS/ASYDE (EPTCS, Vol. 371)*. 1–19. https://doi.org/10.4204/EPTCS.371.1

[2] Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Information and Computation* 75, 2 (1987), 87–106. https://doi.org/10.1016/0890-5401(87)90052-6

[3] Carsten Bormann and Paul E. Hoffman. 2020. Concise Binary Object Representation (CBOR). RFC 8949. , 66 pages. https://doi.org/10.17487/RFC8949

[4] Lesly-Ann Daniel, Erik Poll, and Joeri de Ruiter. 2018. Inferring OpenVPN State Machines Using Protocol State Fuzzing. In *IEEE European Symposium on Security and Privacy (EuroS&P) Workshops* (London, U.K.). IEEE, 11–19. https://doi.org/10.1109/EuroSPW.2018.00009

[5] Joeri de Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *24th USENIX Security Symposium* (Washington, D.C., USA). USENIX Association, 193–206. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter

[6] Tiago Ferreira, Harrison Brewton, Loris D'Antoni, and Alexandra Silva. 2021. Prognosis: Closed-box Analysis of Network Protocol Implementations. In *ACM SIGCOMM Conference*. ACM, 762–774. https://doi.org/10.1145/3452296.3472938

[7] Paul Fiterau-Brostean, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. 2022. DTLS-Fuzzer: A DTLS Protocol State Fuzzer. In *15th IEEE Conference on Software Testing, Verification and Validation* (Valencia, Spain) *(ICST 2022)*. IEEE, 456–458. https://doi.org/10.1109/ICST53961.2022.00051

[8] Paul Fiterau-Brostean, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. 2023. Automata-Based Automated Detection of State Machine Bugs in Protocol Implementations. In *Network and Distributed System Security Symposium* (San Diego, CA, USA) *(NDSS 2023)*. The Internet Society. https://www.ndss-symposium.org/wp-content/uploads/2023/02/ndss2023_s68_paper.pdf

[9] Paul Fiterău-Broștean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. 2020. Analysis of DTLS Implementations Using Protocol State Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2523–2540. https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean

[10] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2015. PULSAR: Stateful Black-Box Fuzzing of Proprietary Network Protocols. In *Security and Privacy in Communication Networks (LNICST, Vol. 164)*. Springer, Cham, 330–347. https://doi.org/10.1007/978-3-319-28865-9_18

[11] Serge Gorbunov and Arnold Rosenbloom. 2010. AutoFuzz: Automated Network Protocol Fuzzing Framework. *International Journal of Computer Science and Network Security (IJCSNS)* 10, 8 (Aug. 2010), 239–245.

[12] Malte Isberner, Falk Howar, and Bernhard Steffen. 2015. The Open-Source LearnLib - A Framework for Active Automata Learning. In *Computer Aided Verification - 27th International Conference, CAV (LNCS, Vol. 9206)*. Springer, 487–495. https://doi.org/10.1007/978-3-319-21690-4_32

[13] Roberto Natella. 2022. StateAFL: Greybox fuzzing for stateful network servers. *Empir. Softw. Eng.* 27, 7 (Oct. 2022). https://doi.org/10.1007/s10664-022-10233-3

[14] Van-Thuam Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNET: A Greybox Fuzzer for Network Protocols. In *IEEE 13th International Conference on Software Testing, Validation and Verification (ICST 2020)*. IEEE, 460–465. https://doi.org/10.1109/ICST46399.2020.00062

[15] Jim Schaad. 2022. CBOR Object Signing and Encryption (COSE): Structures and Process. RFC 9052. , 66 pages. https://doi.org/10.17487/RFC9052

[16] Göran Selander, John Preuß Mattsson, and Francesca Palombini. 2022. Ephemeral Diffie-Hellman Over COSE (EDHOC). draft-ietf-lake-edhoc-18. , 99 pages. https://datatracker.ietf.org/doc/draft-ietf-lake-edhoc/18/

[17] Göran Selander, John Preuß Mattsson, Francesca Palombini, and Ludwig Seitz. 2019. Object Security for Constrained RESTful Environments (OSCORE). RFC 8613. , 94 pages. https://doi.org/10.17487/RFC8613

[18] Zach Shelby, Klaus Hartke, and Carsten Bormann. 2014. The Constrained Application Protocol (CoAP). RFC 7252. https://doi.org/10.17487/RFC7252

[19] Juraj Somorovsky. 2016. Systematic Fuzzing and Testing of TLS Libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. ACM, New York, NY, USA, 1492–1504. https://doi.org/10.1145/2976749.2978411

[20] Bernhard Steffen, Falk Howar, and Maik Merten. 2011. *Introduction to Active Automata Learning from a Practical Perspective*. Springer, Berlin, Heidelberg, 256–296. https://doi.org/10.1007/978-3-642-21455-4_8

[21] Frits W. Vaandrager. 2017. Model learning. *Commun. ACM* 60, 2 (2017), 86–95. https://doi.org/10.1145/2967606

Received 2023-05-18; accepted 2023-06-08