SMBugFinder: An Automated Framework for Testing Protocol Implementations for State Machine Bugs

Paul Fiterău-Broştean
Bengt Jonsson
paul.fiterau_brostean@it.uu.se
bengt@it.uu.se
Uppsala University
Uppsala, Sweden

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

Fredrik Tåquist
fredrik.takvist@it.uu.se
Uppsala University
Uppsala, Sweden

ABSTRACT

Implementations of stateful network protocols must keep track of the presence, order and type of exchanged messages. Any errors, so-called state machine bugs, can compromise security. SMBugFinder provides an automated framework for detecting these bugs in network protocol implementations using black-box testing. It takes as input a state machine model of the protocol implementation which is tested and a catalogue of bug patterns for the protocol conveniently specified as finite automata. It then produces sequences that expose the catalogued bugs in the tested implementation. Connection to a harness allows SMBugFinder to validate these sequences. The technique behind SMBugFinder has been evaluated successfully on DTLS and SSH in prior work. In this paper, we provide a user-level view of the tool using the EDHOC protocol as an example.

CCS CONCEPTS

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

KEYWORDS

Software security, protocol security, network protocols, protocol state fuzzing, model-based testing, model checking

ACM Reference Format:
Paul Fiterău-Broştean, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. 2024. SMBugFinder: An Automated Framework for Testing Protocol Implementations for State Machine Bugs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3650212.3685310

1 INTRODUCTION

Many applications heavily rely on network protocols being properly implemented. Implementations of stateful network protocols such as SSH, TCP, TLS, QUIC, etc. must maintain a state machine by which they keep track of the presence, order and type of the messages exchanged. Flaws in this state machine, so-called *state machine bugs*, can render implementations vulnerable to attacks, e.g., establishing a connection with a server without providing all the credentials, . *Protocol state fuzzing* is a technique that has shown to be very effective at detecting these bugs in implementations of protocols such as TLS [3], DTLS [8], QUIC [5] and SSH [10]. Protocol state fuzzing uses model learning to extract an (approximate) model of the tested implementation's state machine, which is then analyzed for bugs. Analysis of state machines has often been done manually [3, 8], by careful inspection of the model. Recently, it also has been done automatically using model checking [6, 10].

SMBugFinder is a cross-platform protocol-agnostic testing tool whose aim is to make detection and exposition of state machine bugs a fully automated process. It takes as input a (possibly learned) model of a protocol implementation under test (SUT) and a catalogue of automata-encoded patterns for state machine bugs that the SUT is checked for. It uses the patterns to detect and, with provision of a test harness, validate sequences exposing these bugs in the model. Successfully validated sequences are turned automatically into executable test cases. SMBugFinder can compliment existing protocol learning tools, such as DTLS-Fuzzer [9] and EDHOC-Fuzzer [15]. These tools can generate the state machines for SUTs, which SMBugFinder can then automatically analyze using the test harnesses built into these tools for validation.

In prior work published at NDSS'23 [6], an earlier version of SMBugFinder was used to detect a significant number of bugs and security vulnerabilities in a wide range of different DTLS and SSH implementations. In this tool demo paper, we provide a user-level view of the tool and explain its functionality using a more lightweight and more recent network security protocol (EDHOC).

2 BACKGROUND

2.1 EDHOC

Ephemeral Diffie-Hellman over COSE (EDHOC) is a lightweight, authenticated key exchange protocol, designed to achieve security in highly constrained settings. Most notably, EDHOC enables endpoints to communicate securely over Constrained Application Protocol (CoAP), which is an adaptation of HTTP for resource constrained devices. In this scenario, CoAP peers employ EDHOC to establish a security context (i.e., cryptographic keys) for Object Security for Constrained RESTful Environments (OSCORE), which is a lightweight protocol securing communication over CoAP.

Peers performing the EDHOC key exchange can have one of two roles: *initiator* or *responder*. These roles are not tied to the underlying transfer protocol used. For instance, when CoAP is used, a CoAP client can be either an initiator or a responder for the EDHOC

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3685310

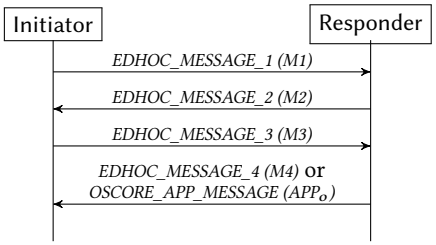


Figure 1: Sequence diagram of an EDHOC key exchange between CoAP nodes. We also show shorthands for messages.

protocol. An EDHOC interaction (Fig. 1) starts with the two peers exchanging *EDHOC_MESSAGE_1 (M1)* and *EDHOC_MESSAGE_2 (M2)* by which they communicate their respective Diffie-Hellman parameters for key exchange and negotiate the algorithms for encryption/authentication. By exchanging *M2* and *EDHOC_MESSAGE_3 (M3)*, the two sides mutually authenticate and establish session keys. The key exchange ends with the responder sending either an *EDHOC_MESSAGE_4 (M4)* or an OSCORE-protected application message *OSCORE_APP_MESSAGE (APP$_O$)*.

2.2 Model Learning

Active automata learning, or *model learning* [20] for short, is an automated black-box testing technique which, by systematically querying a SUT, infers a state machine model, traditionally a Mealy machine, capturing the SUT's input/output behavior. Model learning needs the SUT's input and output alphabets. Additionally, some abstraction is often required in order to interact with the SUT, i.e., a way to map concrete protocol messages to abstract alphabet symbols and vice versa. In practice, abstraction is implemented in a component known as a *test harness* or *mapper*. It is important to note that the models generated by model learning are in most cases approximate, and may not always accurately reflect the SUT's actual behavior. Hence, any flaws found in the model must be validated on the SUT by appropriate tests.

3 EXAMPLE OF SMBUGFINDER'S USAGE

As demonstration, we will show how to employ SMBugFinder to test the EDHOC initiator implementation incorporated in Californium (commit f994359), a widely-used platform for Internet of Things. We will refer to this SUT implementation as RISE, after its developers' affiliation. Protocol state fuzzing has been used to test this very implementation, relying on manual analysis of the model [15]. Here we aim to do the same, but with analysis that is fully automated. This requires first producing the inputs that SMBugFinder needs for automated testing: the SUT model (§3.1) and the bug patterns for the protocol (§3.2) which we assemble into a catalogue (§3.3). We then deploy and run SMBugFinder on these inputs (§3.4). SMBugFinder will produce results which we can analyze (§3.5). We go over these exact steps in our video demonstration.

3.1 Generating a SUT Model

We can generate a model of the RISE SUT by employing EDHOC-Fuzzer [15], using an input alphabet including all EDHOC messages in Fig. 1 plus the *EDHOC_ERROR_MESSAGE* (*ERR$_E$*), which EDHOC uses to signal errors. After some minutes, EDHOC-Fuzzer produces

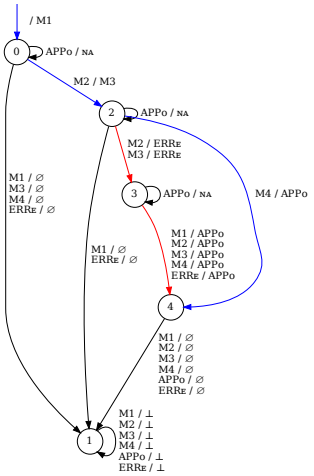


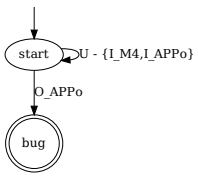
Figure 2: Reduced model of a RISE initiator implementation.

the Mealy machine model in DOT format, ready to be used as-is by SMBugFinder. Note that this DOT format is common across all protocol state learning tools we know of, meaning that SMBugFinder is able to work with the models that all these tools produce. Figure 2 provides a visualization of the model, which is automatically generated using the `dot` utility in Graphviz. For illustration purposes, we have manually added the input *M1* on an incoming edge to the initial state `0`,[1] and we have colored edges on the model that are relevant for its analysis.

Inspecting this model, one can see the blue path $0 \longrightarrow 2 \longrightarrow 4$ of the regular key exchange. Paths traversing also red edges relate to a vulnerability and a state machine bug, identified in prior works manually [15]. We will next see how we can supply bug patterns that enable SMBugFinder to detect these flaws automatically.

3.2 Encoding Bugs as Automata

Paths from state 0 to state 4 traversing red-colored edges, such as $0 \longrightarrow 2 \longrightarrow 3 \longrightarrow 4$, expose a severe vulnerability; on three transitions connecting state 3 to state 4 the responder SUT seemingly completes the key exchange by generating the OSCORE-protected message APP_O, despite not having received *M4* or APP_O. This violates the EDHOC specification [16, §5.4.2] and is, in fact, a vulnerability, as it exposes the OSCORE context, which the two sides can derive after *M3* is received. The bug pattern we use to detect this vulnerability is shown below, alongside the source code implementing it. Like our SUT model, the bug pattern is also in DOT format and can be viewed with the `dot` utility. Code used to specify the initial state (state `start`) is displayed in gray.

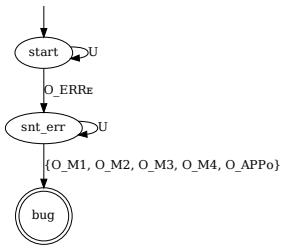



```
digraph G {
  bug [shape="doublecircle"]
  start -> start [label="U - {I_M4, I_APPo}"]
  start -> bug [label="O_APPo"]
  __start0 [label="" shape="none"
            width="0" height="0"]
  __start0 -> start
}
```

[1] Message *M1* is generated before each learning query and, as a result, does not appear in the learned models.

The above pattern automaton captures (accepts) all sequences which end in an initiator-produced APP_O message, without including *M4* or APP_O messages from the responder. The pattern encodes a deterministic finite automaton (DFA) that is specified *for all* the SUT model's input and output symbols, which appear in the pattern prefixed by *I_* and *O_*, respectively. Yet to facilitate pattern specification, encoding is done in a compact way, using the modeling language that our earlier work presented [6]. Specifically, on an edge outgoing from a state, the symbol *U* represents the set of SUT symbols which are not included in an outgoing edge from that state. In our example, on edge `start` → `start`, *U* includes all EDHOC symbols except O_APP_O (i.e., *I_M1*, *I_M2*, etc.). Edges can also contain set expressions, such as our subtraction of the set of symbols *I_M4* and I_APP_O from *U*. Finally, symbols for which no transition is specified (e.g., *I_M4* in state `start` or all symbols in state `bug`) lead to an implicit rejecting sink state. The compactness of the encoding is critical to the usability of SMBugFinder.

The edge 2 → 3 reveals another problem, which is that after generating ERR_E, the SUT does not abort the session as mandated by the EDHOC specification [16, §6]. To capture such sequences on the SUT model, we use the bug pattern below. Note that, after seeing ERR_E from the initiator, the pattern transitions to `snt_err` from which it transitions to `bug` on any output indicating that the SUT has not aborted the session.




```
digraph G {
  bug [shape="doublecircle"]
  start -> start [label="U"]
  start -> snt_err [label="O_ERRE"]
  snt_err -> snt_err [label="U"]
  snt_err -> bug [label=
  "{O_M1, O_M2, O_M3, O_M4, O_APPo}"]
  __start0 [label="" shape="none"
            width="0" height="0"]
  __start0 -> start
}
```

3.3 Assembling a Catalogue of Bug Patterns

Bug patterns are provided to SMBugFinder via a bug catalogue file formatted in XML. This file contains for each bug considered, a path to the pattern DOT model along with metadata on the bug, such as a description and (optionally) severity, which will be shown to the user if the bug is detected. Given below is an excerpt of the catalogue used for our demonstration.

```
<bugPatterns>
 ...
 <bugPattern>
  <name>Initiator completes EDHOC without M4 or APPo</name>
  <bugLanguage>../../common/I_without_m4_appo.dot</bugLanguage>
  <description>Initiator...</description>
  <severity>HIGH</severity>
 </bugPattern>
 ...
<bugPatterns>
```

3.4 Deploying and Running SMBugFinder

SMBugFinder's main dependencies are the development kit for Java and Maven. After installation, bug detection can be performed in the few commands shown below run from SMBugFinder's home directory. Commands and arguments in blue enable validation.

```
1  $ mvn install
2  $ ${launch_test_harness}
3  $ java -jar target\smbugfinder.jar \
4      -m ${path_to_sut_model_dot} -c ${path_to_catalogue_xml} \
5      -vb -ha ${test_harness_address}
6  $ more output/bug_report.txt
```

Without Validation. First we use Maven to install SMBugFinder, which produces an executable JAR file (line 1). Through this file, we invoke the tool using the model (provided by `-m` in line 3) and bug catalogue (provided by `-c` in line 4) we created earlier. Finally, we can open the generated bug report to analyze the results (line 6).

With Validation. Validation, enabled by the option `-vb`, requires an external test harness (launched in line 2). SMBugFinder communicates with this harness over TCP sockets, sending to it inputs to execute on the SUT, and for each input, receiving the SUT's response. Suitable test harnesses are built into many protocol learning tools. In some cases (e.g., the learning tool for SSH [10]) these already support the feature of being controlled over sockets. Where not, this feature can be added by a simple extension, which we have done for our EDHOC demonstration.

Scalability. In our experience, provided an accurate SUT model, SMBugFinder completes detection quickly, in less than one minute, even when the model is large or contains many bugs. Without validation, detection finishes in a matter of seconds. SMBugFinder has been shown to handle SUT models with hundreds of states [6].

3.5 Analyzing SMBugFinder's Results

SMBugFinder produces an output folder containing a bug report document, which provides information on every bug detected on the SUT model. This includes the name and other information about the bug, as specified in the bug catalogue, validation status, and, for validated bugs, the input sequence which can expose the bug on the SUT itself. The output directory also contains intermediary results of bug detection, such as, for each bug, the DOT model of its DFA in expanded form as well as DFAs capturing sequences exposing the bug on the SUT model. Included are also test cases for validated bugs, or for bugs found in the SUT model when validation was not enabled. Below, we show the contents of the test case file SMBugFinder generated for the RISE initiator, exposing the vulnerability discussed in §3.2. The test case, comprising line-separated alternating input/output sequences, can be executed as-is by test harnesses from learning tools built on ProtocolState-Fuzzer, which include DTLS-Fuzzer [9] and EDHOC-Fuzzer [15].

```
EDHOC_MESSAGE_2
#EDHOC_MESSAGE_3
EDHOC_MESSAGE_2
#EDHOC_ERROR_MESSAGE
EDHOC_MESSAGE_2
#OSCORE_APP_MESSAGE
```

4 IMPLEMENTATION

SMBugFinder is cross-platform, entirely built in Java. Architecturally, SMBugFinder is split into two major components; cf. Fig. 3. `BugPatternLoader` is responsible for parsing the bug catalogue and

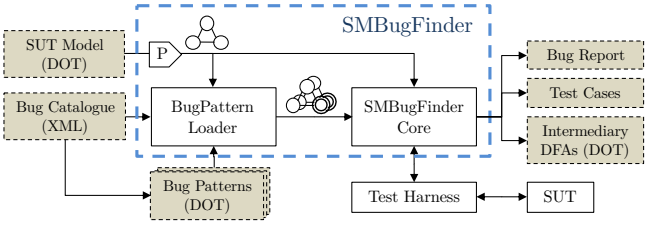


Figure 3: SMBugFinder's Architecture

the associated bug patterns. The parsed patterns are expanded during loading to form fully-specified DFAs using the alphabet of symbols extracted from the parsed SUT model. They are packed together with the bug metadata and provided to the `SMBugFinderCore`.

`SMBugFinderCore` uses the received automata for the implementation under test and bug patterns to detect bugs in the SUT, following the procedures described in our earlier work [6]. Specifically, it first automatically translates the SUT Mealy machine to a DFA. For each bug pattern, it then computes the intersection of the pattern DFA with the SUT DFA. Whenever the resulting DFA is not empty, it generates from it sequences which expose the bug on the SUT model. If validation is enabled, these sequences are then executed on the SUT via a test harness until one of them is successfully validated or the set of sequences is exhausted.

SMBugFinder relies heavily on the automata library automatalib, using its automata classes to store SUT and bug pattern automata, its functionality for parsing automata from DOT files, and its built-in support for efficiently performing operations on automata including intersection. For parsing bug pattern edges, SMBugFinder uses a parser generated via JavaCC on the basis of a grammar file.

Generic Library. SMBugFinder can be used both as a standalone testing tool or as a library, which can be incorporated, e.g., into an existing protocol learning tool to enable automated detection. In its library capacity, SMBugFinder provides the following interfaces (Fig. 4), which tool developers need to provide implementations for.

- `SymbolicMapping` is an interface for mapping (abstracted) input/output symbols of the SUT's Mealy machine to SMBugFinder's internal symbol representation, which is used in bug pattern and SUT DFAs. SMBugFinder includes a basic implementation for when the SUT symbols are strings.
- `SUT` is an interface that a test harness should implement, for executing selected input words (i.e., sequences) on the SUT and retrieving the resulting output words. SMBugFinder includes an implementation which communicates over sockets with an external test harness.

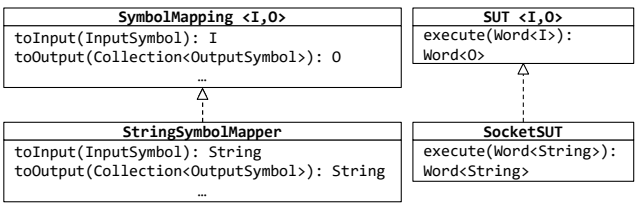


Figure 4: SMBugFinder's main interfaces and included implementations.

5 RELATED WORK

Protocol state fuzzing has been applied to test implementations of major network protocols, several (e.g., TLS [3] and DTLS [8]) using manual analysis of the model. Some works (e.g., on TCP [7], SSH [10] and IPsec [12]), have automated analysis by encoding properties in linear temporal logic (LTL) and using model checking to assess if they are satisfied on the learned model. By comparison, we encode bugs as finite automata, which strictly are more expressive than LTL for encoding safety properties.

Another related approach employs *differential testing*, whereby model learning is used to generate models of multiple implementations, which are then compared to detect discrepancies. This approach has been applied to TCP [1], MQTT [17] and 4G LTE [13]. Relevant for these approaches and ours are advancements in model learning including algorithms that require fewer queries to learn a SUT, such as TTT [14] and more recently $L^{\#}$ [19], or that can generate richer models, such as SL^{*} [2] and more recently SL^{λ} [4]. Among related tools we mention protocol learning tools such DTLS-Fuzzer [9], EDHOC-Fuzzer [15] and Prognosis [5] (which supports TCP and QUIC). We also note TLS-Attacker [18], an extensive testing framework for (D)TLS.

6 TOOL AVAILABILITY

SMBugFinder is open source and available on a GitHub repository. A video demonstrating the tool can also be found there. The source code of the commit at the time of this writing is also archived [11].

SMBugFinder's documentation includes a `README.md` which details how to install SMBugFinder, apply it for SSH, and interpret the results. It explains the inputs SMBugFinder uses, and the provision of such inputs for the DTLS v1.2, SSH, and EDHOC protocols. Bug patterns for other network protocols can be added in similar ways.

7 FUTURE IMPROVEMENTS

In this paper, we overviewed SMBugFinder, an automated framework for testing protocol implementations for state machine bugs. SMBugFinder is fully automatic, treats implementations as black box and has shown to be applicable to several protocols [6].

Our current plan is to extend SMBugFinder as follows:

- *Extended Protocol Support:* The effectiveness of SMBugFinder is determined by the extensiveness of its bug patterns. As mentioned, currently SMBugFinder comes with bug pattern catalogues for DTLS, SSH, and EDHOC. We want to extend this collection, by adding patterns for other protocols.
- *Richer Bug Patterns:* Newer model learner algorithms and their implementations (such as SL^{λ} [4] implemented in RALib) can generate extended finite state machines. These are richer than Mealy machines, and thus, are able to capture more bugs, e.g., bugs in an implementation's handling of data parameters. A direction is to make SMBugFinder work with these richer models.

ACKNOWLEDGMENTS

This work was partially funded by the Swedish Research Council, the Swedish Foundation for Strategic Research's project aSSIsT, and the Knut and Alice Wallenberg Foundation's project UPDATE.

REFERENCES

[1] George Argyros, Ioannis Stais, Suman Jana, Angelos D. Keromytis, and Aggelos Kiayias. 2016. SFADiff: Automated Evasion Attacks and Fingerprinting Using Black-box Differential Automata Learning. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. ACM, 1690–1701. https://doi.org/10.1145/2976749.2978383

[2] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. 2016. Active learning for extended finite state machines. *Formal Aspects of Computing* 28, 2 (01 April 2016), 233–263. https://doi.org/10.1007/s00165-016-0355-5

[3] 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

[4] Simon Dierl, Paul Fiterau-Brostean, Falk Howar, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. 2024. Scalable Tree-based Register Automata Learning. In *Tools and Algorithms for the Construction and Analysis of Systems (LNCS, Vol. 14571)*, Bernd Finkbeiner and Laura Kovács (Eds.). Springer Nature Switzerland, Cham, 87–108. https://doi.org/10.1007/978-3-031-57249-4_5

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

[6] Paul Fiterau-Brostean, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. 2023. Automata-Based Automated Detection of State Machine Bugs in Protocol Implementations. In *30th Annual Network and Distributed System Security Symposium* (San Diego, CA, USA) *(NDSS 2023)*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/automata-based-automated-detection-of-state-machine-bugs-in-protocol-implementations/

[7] Paul Fiterău-Broştean, Ramon Janssen, and Frits W. Vaandrager. 2016. Combining Model Learning and Model Checking to Analyze TCP Implementations. In *Computer Aided Verification - 28th International Conference, CAV 2016, Proceedings, Part II (LNCS, Vol. 9780)*. Springer, 454–471. https://doi.org/10.1007/978-3-319-41540-6_25

[8] Paul Fiterău-Broştean, 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

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

[10] Paul Fiterău-Broştean, Toon Lenaerts, Joeri de Ruiter, Erik Poll, Frits W. Vaandrager, and Patrick Verleg. 2017. Model Learning and Model Checking of SSH Implementations. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software* (Santa Barbara, CA, USA) *(SPIN 2017)*. ACM, 142–151. https://doi.org/10.1145/3092282.3092289

[11] Paul Fiterău-Broştean, Konstantinos Sagonas, Fredrik Tåquist, and Bengt Jonsson. 2024. SMBugFinder: An Automated Framework for Testing Protocol Implementations for State Machine Bugs. https://doi.org/10.5281/zenodo.12665353 Artifact for the ISSTA '24 paper with the same title.

[12] Jiaxing Guo, Chunxiang Gu, Xi Chen, and Fushan Wei. 2019. Model Learning and Model Checking of IPSec Implementations for Internet of Things. *IEEE Access* 7 (Nov. 2019), 171322–171332. https://doi.org/10.1109/ACCESS.2019.2956062

[13] Syed Rafiul Hussain, Imtiaz Karim, Abdullah Al Ishtiaq, Omar Chowdhury, and Elisa Bertino. 2021. Noncompliance as Deviant Behavior: An Automated Black-box Noncompliance Checker for 4G LTE Cellular Devices. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*. ACM, 1082–1099. https://doi.org/10.1145/3460120.3485388

[14] Malte Isberner, Falk Howar, and Bernhard Steffen. 2014. The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning. In *Runtime Verification: 5th International Conference, RV 2014, Proceedings*. LNCS, Vol. 8734. Springer, 307–322. https://doi.org/10.1007/978-3-319-11164-3_26

[15] 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* (Seattle, WA, USA) *(ISSTA '23)*, René Just and Gordon Fraser (Eds.). ACM, New York, NY, USA, 1495–1498. https://doi.org/10.1145/3597926.3604922

[16] Göran Selander, John Preuß Mattsson, and Francesca Palombini. 2024. Ephemeral Diffie-Hellman Over COSE (EDHOC). RFC 9528. https://doi.org/10.17487/RFC9528

[17] Martin Tappler, Bernhard K. Aichernig, and Roderick Bloem. 2017. Model-Based Testing IoT Communication via Active Automata Learning. In *IEEE International Conference on Software Testing, Verification and Validation* (Tokyo, Japan) *(ICST 2017)*. IEEE Computer Society, 276–287. https://doi.org/10.1109/ICST.2017.32

[18] TLS-Attacker Online; accessed 31-July-2024. TLS-Attacker. https://github.com/tls-attacker/TLS-Attacker.

[19] Frits Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. 2022. A New Approach for Active Automata Learning Based on Apartness. In *Tools and Algorithms for the Construction and Analysis of Systems (LNCS, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 223–243. https://doi.org/10.1007/978-3-030-99524-9_12

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

Received 2024-07-05; accepted 2024-07-26