Stateful Fuzzing of OPC UA

Cristian Daniele, Mark Fijneman, and Erik Poll

Radboud University, Nijmegen, the Netherlands
{cristian.daniele,mark.fijneman,erik.poll}@ru.nl

Abstract. Fuzzing is a widely used and effective testing technique used to discover bugs in systems. Fuzzers like AFL++ (and its descendants) daily find plenty of bugs in open-source implementations. Unfortunately, the statefulness of some systems (such as protocol implementations) limits the effectiveness of stateless fuzzers since they cannot dig deep into the state model. For such stateful systems, it is necessary to take states into account to get the best results.
For example, in recent years, the research community has shown great interest in the security of the OPC UA, a protocol used in Industrial Control Systems. Although a few bugs were found during previous analyses of the protocol, they did not consider the statefulness of the protocol.
In this paper, we fuzz three OPC UA implementations with three different fuzzers (AFLnwe, BooFuzz and AFLNet) and find three novel bugs overlooked in the previous analyses.
Our results highlight the importance of considering the systems' stateful nature when choosing the fuzzer to use. Also, they emphasise the importance of having complete grammar when opting for grammar-based fuzzers, i.e., fuzzers that need the messages and state model specification in input.

Keywords: Protocol testing · Stateful fuzzing · OPC UA.

1 Introduction

During the last few years, the fuzzing community highlighted the need for fuzzers specifically tailored to stateful systems [8]. This need is demonstrated by the plethora of stateful fuzzers the community developed to tackle stateful software of different natures (e.g., network [19], embedded [18,11], or IoT protocols [10]). Unfortunately, security analysts often know very little about the nature of the systems they are testing and about the state model they might implement. Also, the lack of documentation and the incomplete protocol specification can lead security analysts to make the wrong assumption about the statefulness of the system.

In this paper, we fuzz three OPC UA implementations by using the stateless fuzzer AFLnwe[1], the grammar-based fuzzer BooFuzz [14] and the stateful fuzzer AFLNet [15]. The contribution of this paper is twofold:

[1] `https://github.com/thuanpv/aflnwe`

1. we show that wrong assumptions about the stateful nature of a system can lead to misreading its robustness. In fact, the previous efforts [6,5,9] overlooked the stateful nature of OPC UA, thereby missing three novel bugs;
2. as a consequence, we prove that stateless fuzzers achieve poor results when testing the robustness of stateful systems. Moreover, we show how the effectiveness of a grammar-based fuzzer strongly depends on the quality of the grammar provided in input.

The paper is structured as follows: Section 2 introduces the background by presenting an overview of fuzzing and the OPC UA protocol. Section 3 presents the earlier efforts. In Section 4, we fuzz three OPC UA implementations with AFLnwe, BooFuzz and AFLNet. In Section 5, we discuss the results, in Section 6, we present the future work and in Section 7 we conclude by highlighting the contribution of the paper.

2 Background

2.1 Stateless and Stateful Fuzzing

Fuzzing is a testing technique widely used to find vulnerabilities in software. The idea behind fuzzing is straightforward: the fuzzer sends many malformed messages to the Software Under Test (SUT) waiting for crashes or unexpected system behaviours. Even though fuzzing is very effective in testing stateless systems, it does not always work well when dealing with the stateful ones. With the term *stateful systems*, we mean all those systems that need an internal state to work correctly. In those systems, the response to a certain message m_i is not exclusively influenced by m_i, but also by the former messages $\{m_{i-1}, m_{i-2}, \dots\}$. For this reason, while stateless fuzzers only have to mutate single messages (by mutating, for example, the message m_i into the message m_i^1); stateful fuzzers also need to mutate the message order (often called *trace*). For example, a valid — and perhaps interesting — mutation over the trace $t_0 = \{m_1, m_2, m_3\}$ is the trace $t_1 = \{m_1, m_3, m_2\}$.

Among the other techniques for stateful fuzzing [8], two stood out in the last years: grammar-based and coverage-guided fuzzing.

Grammar-based fuzzers require the definition of the structure of the messages and the structure of the state model. Also, it is possible to give the fuzzer information about the *most interesting* states – in terms of chances of triggering bugs – to fuzz. Despite this approach being originally devised to fuzz stateless systems, it works also with the stateful ones. Stateful systems usually rely on specific protocols and message specifications that can be expressed through grammar. In other words, the only difference between stateless and stateful grammar-based fuzzers relies on the grammar used. The former only use the grammar of the message, the latter also the grammar of the traces.

One of the limits of this approach is that the grammar provided has to be complete since it has to cover all the message fields and system states. An example of a stateful grammar-based fuzzer is Boofuzz [14], which we use to fuzz three OPC UA implementations (Section 4.2).

Coverage-guided fuzzers, also known as stateful grey-box fuzzers, do not require a grammar in input but, instead, use *seed traces* and *feedback* from the system to infer and explore the state model of the SUT and generate interesting traces. More in detail, they start their fuzzing campaign with a set of seed traces they will then mutate. To increase the probability of generating interesting traces, they use feedback from the system to steer the generation of new inputs. An example of a stateful coverage-guided fuzzer is AFLNet [15], which we used in our experiments (Section 4.3).

The majority of the (stateless and stateful) coverage-guided fuzzers inject instrumentation into the SUT to generate test cases that, with a high probability, will cover new parts of the code.

Nevertheless, conventional (stateless) coverage-guided fuzzers usually have a hard time fuzzing stateful systems as they do not have the knowledge to browse around the state model. An example of a stateless fuzzer able is AFLnwe, which we used to test three OPC UA implementations (Section 4.1).

2.2 OPC UA

OPC UA is a stateful protocol used for the communication between Industrial Control Systems (ICS). The protocol requires the client to open a secure channel (or an authenticated session) and discover *service sets*, i.e. a group of messages that are used to fulfil one purpose within an OPC UA connection. In order to evaluate the state coverage of the fuzzers, we inferred the OPC UA state model by hand by looking at the specification [2] and by observing OPC UA network traffic (for the three implementations) between client and server (Figure 1).

In this paper, we focus on the commands (called *message types* in the specification) that are responsible for establishing connections with an OPC UA server. These commands are included within a service set. For example, the commands service sets *SecureChannel*, *Discovery*, and *Session* are required to establish a connection between a client and a server.

More in detail, we pay attention to nine commands (or message types), spread out in four service sets.

- *Basic message types*
  1. Hello
- *SecureChannel service set*
  2. OpenSecureChannel
  3. CloseSecureChannel
- *Discovery service set*
  4. FindServers
  5. FindServersOnNetwork
  6. RegisterServer
  7. GetEndpoints
- *Session service set*
  8. CreateSession
  9. ActivateSession

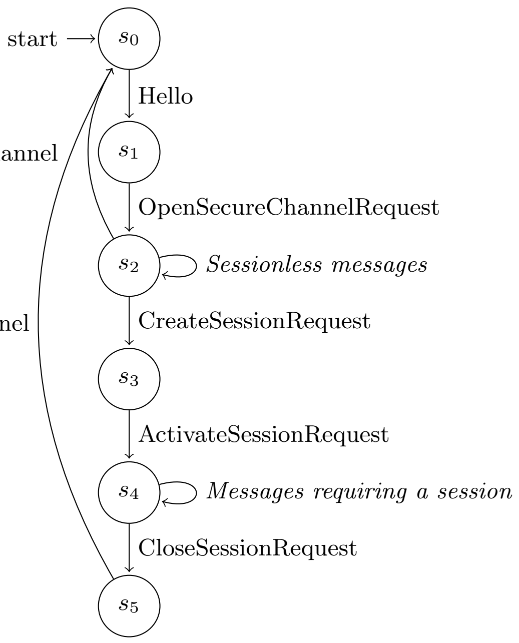


Fig. 1: Manual reconstruction of the OPC UA state model

3 Previous Fuzzing Efforts

In the last years, different organizations have shown interest in testing the robustness of OPC UA both on the implementation [6,5,9] and protocol level [13,17]. Despite the considerable effort, all the attempts made to test the robustness of the protocol implementations overlooked the stateful nature of OPC UA. This wrong assumption precluded the possibility of finding bugs that rely deeply on the state model (Section 5.1),which we found with the stateful fuzzer AFLNet.

3.1 German Federal Office for Information Security (BSI)

In 2017 the German Federal Office for Information Security, also known as BSI, published an extensive security analysis of OPC UA where they analysed the ANSI C implementation [6]. They tested the implementation with the fuzzer *Peach* [3] but no crashes were found.

In 2022, they tested open62541 by using the black-box fuzzer BooFuzz[2] and the coverage-guided fuzzer[3] libFuzzer [9], both with ASan[4]. The latter fuzzing campaign found several crashes, including a heap buffer overflow in an old open62541 implementation.

[2] The code for the fuzzing campaigns is on GitHub at `https://github.com/fkie-cad/blackbox-opcua-fuzzing`

[3] mistakenly called white-box fuzzing in the report

[4] `https://github.com/google/sanitizers/wiki/addresssanitizer`

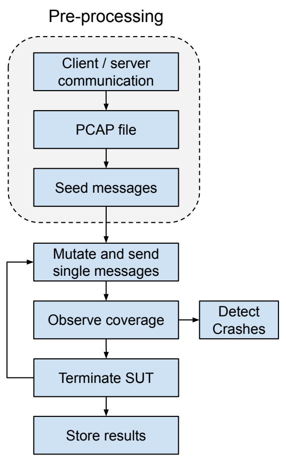


Fig. 2: AFLNwe

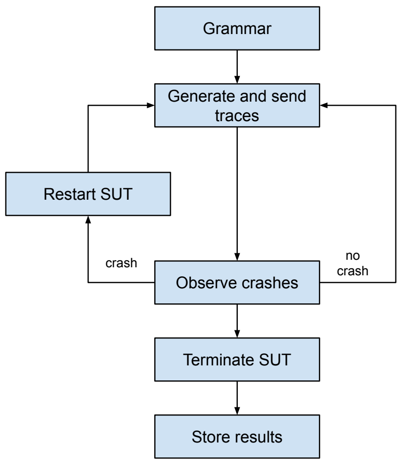


Fig. 3: BooFuzz

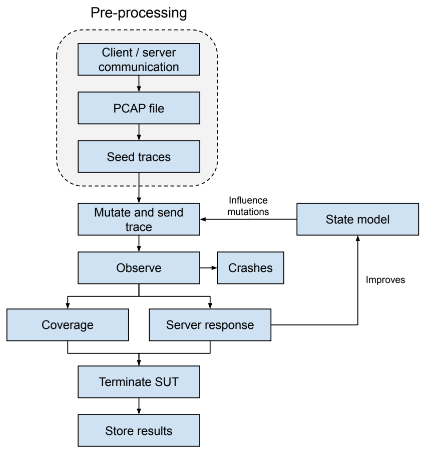


Fig. 4: AFLNet

Fig. 5: Flowcharts of different fuzzers.

3.2 Kaspersky Lab ICS CERT

In 2018, researchers from Kaspersky Lab ICS CERT ran a modified version of AFL (able to deal with network system calls) on the C implementation OPC UA ANSI C implementation and found nine vulnerabilities [5]. However, the ANSI C implementation they tested is no longer maintained and it is not available for download.

4 Fuzzing OPC UA

We fuzzed three OPC UA open source implementations (i.e., open62541, ANSI-C implementation, and FreeOpcUa) with AFLnwe, BooFuzz and AFLNet for 10 hours. We used the ASan in all our experiments. The code is available online.[5]

As summarized in Table 1, AFLnwe only found a bug in the initial state of the protocol, BooFuzz did not find any reproducible bug and AFLNet discovered the same bug found by AFLnwe and two new ones. The three fuzzing campaigns are discussed in more detail below.

4.1 AFLnwe Fuzzing Campaign

Experiments. We tested the OPC UA implementations with AFLnwe, a fork of AFL++ able to send messages over the sockets. AFLnwe — being stateless — does not use any notion of a state model but only takes in input seed messages. We collected such messages by sniffing the communications between a sample

[5] `https://anonymous.4open.science/r/opcua-fuzzing-3743/README.md`

client and a server. The flowchart of the AFLNwe experiments is presented in Figure 2.

Findings. In less than 2 hours, AFLnwe found a bug in the initial state of the FreeOpcUa implementation. The crash is triggered by the server binary terminating due to an uncaught error, stating that the buffer size given to *InputFromBuffer* is invalid. More in detail, error is triggered when the length of the buffer size is 0. In fact, for a coding error, the buffer size can have a value of 0 when both the message and the OPC UA header are 8 bytes long. Despite the project is not actively maintained anymore, we reported the bugs to the developers, who asked us to open a pull request to fix the bug (on which we are currently working).

4.2 BooFuzz Fuzzing Campaign

Experiments. The flowchart of the BooFuzz experiments is presented in Figure 3. The grammar used to fuzz OPC UA covers the following sequence of messages[6]:

- Hello
- Hello → OpenSecureChannelRequest
- Hello → OpenSecureChannelRequest → CloseSecureChannel
- Hello → OpenSecureChannelRequest → FindEndpoints *(Sessionless message)*
- Hello → OpenSecureChannelRequest → GetEndpoints *(Sessionless message)*
- Hello → OpenSecureChannelRequest → FindServersOnNetwork *(Sessionless message)*
- Hello → OpenSecureChannelRequest → RegisterServer2 *(Sessionless message)*
- Hello → OpenSecureChannelRequest → CreateSessionRequest
- Hello → OpenSecureChannelRequest → CreateSessionRequest → ActivateSessionRequest

Please note that such a grammar does not cover $S5$ and does not allow sending messages that require a prior session establishment.

The fuzzer takes the grammar in input and then generates a trace that then forwards to the SUT and monitors potential crashes. If no crash is detected, the fuzzer proceeds with the generation of a new trace. On the contrary, if a crash is triggered, the fuzzer restarts the SUT.

BooFuzz Findings. BooFuzz could not find any bug in any recent version of open62541, FreeOpcUA and ANSI-C implementation with the grammar presented in Section 4.2. The limited state coverage (four states out of five), together

[6] We used the same grammar used by BSI.

	AFLnwe		**BooFuzz**		**AFLNet**		
Implementation	**Unique crashes**	**States covered**	**Unique crashes**	**States covered**	**Unique crashes**	**States covered**	**Runtime**
open62541	0	1/5	0	4/5	0	5/5	∼10 hours
ANSI-C implementation	0	1/5	0	4/5	**1**	5/5	∼10 hours
FreeOpcUa	**1**	1/5	0	4/5	**2**	5/5	∼10 hours

Table 1: Bugs found by AFLnwe, BooFuzz and AFLNet

with the (stateless) mutation strategy of BooFuzz justifies the poor results. Also, Boofuzz — to keep the mutation under control — only mutates the last message of each sequence and not the single messages or their order. In other words, it only uses the grammar as a prefix to reach a certain state that then fuzzes. This approach limits the generation of traces where the order of the messages is changed. Unfortunately, enabling the mutation of the entire trace is not a solution, since — without any feedback from the system — it would produce thousands of malformed traces.

4.3 AFLNet Fuzzing Campaign

Experiments. The flowchart of the AFLNet experiments is presented in Figure 4. As we did for AFLnwe (Section 4.1), we collected some seed traces by sniffing a communication between a client and a server via *tcp dump* (pre-processing).

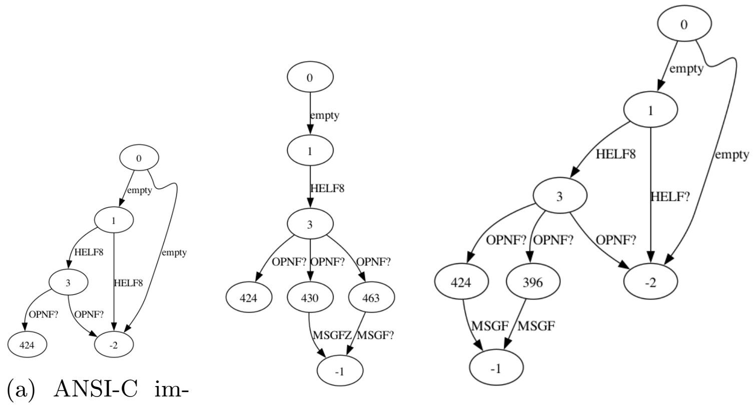
The traffic sniffed allows the fuzzer to reach all the states presented in Figure 1, as it includes the following traces:

– Hello → OpenSecureChannelRequest → FindServers *(Sessionless message)* → CloseSecureChannelRequest
– Hello → OpenSecureChannelRequest → GetEndpoints *(Sessionless message)* → CloseSecureChannelRequest
– Hello → OpenSecureChannelRequest → CreateSessionRequest → ActivateSessionRequest → . . . *(Messages requiring a session)* → CloseSessionRequest → CloseSecureChannelRequest

It is worthwhile to notice that, unlike for BooFuzz, the last captured trace allows fuzzing $S5$ and $S4$ *after* having established a proper session.

AFLNet takes the seed files in input, mutates and forwards them to the SUT. In addition to crashes, the fuzzer observes the feedback on two aspects: the *code coverage* and the *server response*. It uses the information about the coverage to steer the generation of new traces thanks to its grey-box nature. Moreover, it observes the response of the server to infer the state model of the system and use it during the mutation of new traces.

To properly infer the state model, AFLNet requires two *abstraction functions* (one for the requests and one for the responses) to map raw messages into meaningful strings. The *request* abstraction function we implemented takes in input the whole message, reads the *Payload size* and returns the *Payload*. The



(a) ANSI-C implementation. (b) FreeOpcUA. (c) Open62541 state model.

Fig. 6: State models inferred by AFLNet. Edge labels represent the message to trigger the transaction; state labels represent the response code.

response abstraction function simply extracts the response codes from the server responses.

Findings. AFLNet found one crash in the ANSI-C implementation and two crashes in the FreeOPCUa one, as shown in Table 1.

ANSI-C implementation crash. AFLNet found a bug (a segmentation fault) in the ANSI-C implementation. The crash is triggered when sending a *find_ servers* request with a *null* URL. The source of the bug is a string comparison when the first string is null and the second is *"Nano_ Server"*. Using *strncmp* with a null string on either side is considered undefined behaviour, and causes a segmentation fault in this case. However, the bug is present only in the example server provided with the ANSI-C implementation and not the main code of the stack itself. Since this stack is no longer directly supported or available, we did not report this bug.

FreeOpcUa crashes. AFLNet found two unique crashes (Table 1), one of which is the same one discovered by AFLnwe (Section 4.1). The second crash is triggered when a memory limit of 50MB is set. However, new connections are still accepted until a connection limit is hit.

5 Discussions

5.1 Bugs Found

Two of the three bugs found are not shallow, i.e., they cannot be triggered from the first state but need to be triggered from the states $S2$ and $S4$.

This explains why AFLnwe — being stateless — could only find one bug (the shallow one), while AFLNet could also find the others. In fact, AFLNet (like other stateful fuzzers [8]) considers the statefulness of the system and infers — during the fuzzing campaign — the state model of the SUT (as shown in Figure 4 and explained in Section 4.3). Also, its coverage-guided nature helps to prioritise interesting messages and states. In addition, AFLNet performs mutations over the traces and keeps a queue of interesting messages for every state, making the selection of interesting messages state-aware.

On the other hand, Boofuzz did not find any bugs. The poor performances can be attributed to its black-box nature, which does not allow the fuzzer to prioritise interesting messages, and to the inability to perform mutation on the trace level. Also, the grammar used by BooFuzz is not complete (as it does not cover $S5$) and does not allow sending messages that require a prior session establishment (as explained in Section 4.2).

5.2 Limits and Applications of Stateless Fuzzers

All the fuzzers based on AFL — being exclusively devised to tackle stateless systems — restart the SUT after every message. The resets bring the SUT back to the initial state, thereby allowing the fuzzer to send messages only to the shallow state. In other words, the restart stops the fuzzer from exploring deeper states whatsoever. This behaviour prevents AFLnwe from triggering any deep bug. Moreover, stateless fuzzers do not have any notion of state models and, thus, are not capable of focusing on specific states or performing stateful mutations (as explained in Section 2).

Nevertheless, stateless fuzzers might still be used to bypass the statefulness of the systems [7]. For example, fuzzers like libFuzzer [1] allow writing *fuzzing target* to specific functions to bypass the statefulness of some systems. Unfortunately, this solution allows fuzzing only small portions of code and makes bugs triggered by weird message compositions impossible to find. Also, this approach makes the crash not reproducible as the complete trace to trigger the bug is unknown. OSS-fuzz [16] used this approach to fuzz an older version of open62541 and found vulnerabilities, but, unfortunately, it is not possible to trigger those bugs from the outside as the fuzzer only found the last input, i.e. the very last message of the trace that triggered the bug.

Another solution to fuzz stateful systems with stateless fuzzers is cramming multiple messages into one. This would allow the encapsulation of entire traces in one message only, making the restart of the SUT not an issue.

5.3 Limits and Applications of Grammar-Based Fuzzers

Grammar-based fuzzers can be used to fuzz stateful systems. As already mentioned in Section 2, for a grammar-based fuzzer the statefulness of the system does not play an important role as the grammar can describe both the message fields and the state model.

Nevertheless, grammar-based fuzzers struggle in fuzzing stateful systems because of the need for an accurate description of the grammar for both the messages and the traces. Also, having precise grammar would not make black-box grammar-based fuzzers very effective. In fact, they would have to deal with such a large grammar that would make the traces generation random. To mitigate this problem some fuzzers (like BooFuzz, Section 4.2), only mutate the last message of the trace, missing — perhaps — interesting mutations on the trace level.

6 Future Work

As explained in Section 4, we used limited grammar for BooFuzz and non-comprehensive network traffic for AFLNet. Improving such information might improve the code coverage and find new vulnerabilities. Also, we limited our research to open-source OPC UA implementations, but many more exist as has been documented by Erba et al.[12].

We only used network-based fuzzers in both experiments. However, network communication introduces a high amount of latency. To improve fuzzing performance, the SUT could be rewritten to read and write messages through faster channels [4].

7 Conclusions

In this paper, we compare the effectiveness of stateless and stateful fuzzers in fuzzing a stateful system. Table 1 shows that stateless fuzzers achieve poor performance in fuzzing stateful systems because – as expected – they can only find bugs in the initial state. Indeed, AFLnwe only found one shallow bug in the FreeOpcUA implementation.

Grammar-based fuzzers can be used for stateful fuzzing, but only when the analysts provide good grammar. Also, the fuzzer needs to be able to both mutate individual messages and mutate the order of messages. In our experience, BooFuzz does not mutate the order of messages but uses the grammar only to reach specific states.

On the contrary, stateful fuzzers work fine with only seed traces (that are often easy to generate) and provide a fast yet effective assessment of the robustness of the SUT. Indeed, in addition to the bug found by AFLnwe, AFLNet found two other, deeper bugs in two different implementations.

8 Acknowledgement

We thank OMRON for introducing the OPC UA protocol to us. Research funded by NWO as part of the INTERSECT project (NWA.1160.18.301).

References

1. Libfuzzer, `https://llvm.org/docs/LibFuzzer.html`
2. open62541. OPC UA Foundation `https://reference.opcfoundation.org/Core/Part6/v105/docs/7.1.3`

3. Peach fuzzer (2004), `https://gitlab.com/peachtech/peach-fuzzer-community`
4. Andarzian, S.B., Daniele, C., Poll, E.: On the (in) efficiency of fuzzing network protocols. Annals of Telecommunications (2024)
5. Cheremushkin, P., Temnikov, S.: OPC UA security analysis. Kaspersky Lab ICS CERT (2018), `https://ics-cert.kaspersky.com/publications/reports/2018/05/10/opc-ua-security-analysis/`
6. Damm, Gappmeier, Zugfil, Plöb, Fiat, Störtkuhl: OPC UA security analysis. BSI (2017), `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/OPCUA/OPCUA.pdf`
7. Daniele, C.: Is stateful fuzzing really challenging? arXiv preprint arXiv:2406.07071 (2024)
8. Daniele, C., Andarzian, S.B., Poll, E.: Fuzzers for stateful systems: Survey and research directions. ACM Computing Surveys **56**(9), 1–23 (2024)
9. vom Dorp, J., Merschjohann, S., Meier, D., Patzer, F., Karch, M., Haas, C.: OPC UA security analysis. BSI (2022), `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/OPCUA/OPCUA_2022.html`
10. Eceiza, M., Flores, J.L., Iturbe, M.: Fuzzing the internet of things: A review on the techniques and challenges for efficient vulnerability discovery in embedded systems. IEEE Internet of Things Journal pp. 10390–10411 (2021)
11. Eisele, M., Maugeri, M., Shriwas, R., Huth, C., Bella, G.: Embedded fuzzing: a review of challenges, tools, and solutions. Cybersecurity **5**(1), 18 (2022)
12. Erba, A., Müller, A., Tippenhauer, N.O.: Security analysis of vendor implementations of the OPC UA protocol for industrial control systems. In: Proceedings of the 4th Workshop on CPS & IoT Security and Privacy. ACM (2022)
13. Mühlbauer, N., Kirdan, E., Pahl, M.O., Carle, G.: Open-source OPC UA security and scalability. In: IEEE International Conference on Emerging Technologies and Factory Automation (ETFA) (2020)
14. Pereyda, J.: Boofuzz, `https://github.com/jtpereyda/boofuzz`
15. Pham, V.T., Böhme, M., Roychoudhury, A.: Aflnet: A greybox fuzzer for network protocols. In: Conference on Software Testing, Validation and Verification (ICST) (2020)
16. Serebryany, K.: OSS-Fuzz - Google's continuous fuzzing service for open source software. USENIX (2017)
17. Tran Van, A., Levillain, O., Debar, H.: Mealy verifier: An automated, exhaustive, and explainable methodology for analyzing state machines in protocol implementations. In: International Conference on Availability, Reliability and Security (2024)
18. Yun, J., Rustamov, F., Kim, J., Shin, Y.: Fuzzing of embedded systems: A survey. ACM Computing Surveys **55**(7), 1–33 (2022)
19. Zhang, Z., Zhang, H., Zhao, J., Yin, Y.: A survey on the development of network protocol fuzzing techniques. Electronics **12**(13), 2904 (2023)