Bachelor's Thesis Computing Science

Radboud University Nijmegen

Fuzzing open source OPC UA implementations

Author:
Mark Fijneman
s1063623

First supervisor/assessor:
Dr. Ir. Erik Poll

Second assessor:
Dr. Stjepan Picek

Second supervisor:
Cristian Daniele

Third supervisor:
Seyed Behnam Andarzian

August 23, 2023

Abstract

A growing number of companies use Industrial Control Systems (ICS) in order to facilitate and speed up production. OPC UA is a widely used standard that describes how devices in ICS should communicate. The functioning of such systems is critical in many cases, and malfunction could cause both economic and physical harm depending on the system, hence it is important that such systems are secured properly.

Therefore, we performed fuzzing of the protocol stack of multiple open-source implementations of the OPC UA standard. Fuzzing is a method of testing a System Under Test (SUT). During a fuzzing test, the fuzzer automatically sends malformed messages to the SUT. The fuzzer continuously checks for crashes during testing. If a crash is found, this could be indicative of a serious flaw that could be used to take over or perform denial of service on a system.

While research has been performed on the security of Open Platform Communications Unified Architecture (OPC UA) protocols, it is either a few years old or lacks depth on the fuzzing part of the research. We performed both black-box and grey-box fuzzing on 7 Open Platform Communications Unified Architecture (OPC UA) implementations and ensured that the experiments were repeatable for future use.

We found that performing grey-box fuzzing using AFLNet was the most effective fuzzing-method. Using this method, we found one crashe in the legacy OPC UA ANSI-C stack and two crashes in FreeOpcUa.

Contents

1 Introduction

Open Platform Communications Unified Architecture (OPC UA) is a standard which is used to exchange messages between different machines within Industrial Control Systems (ICS), such as industrial robots or CNC machines. These messages include device metadata, sensor data and commands. One of the main reasons the OPC UA standard exists is to ensure that communication is possible between different combinations of machines and software. As a result of this, numerous open source implementations of the OPC UA protocol have been written.

In 2020, a study was performed on the security of OPC UA servers that were exposed to the internet [4]. Dahlmanns et al. discovered 1114 accessible OPC UA servers, out of which 92% were configured insecurely. 44% of the found servers allowed unauthenticated users to read, write and/or execute other commands. The fact that a high amount of these systems are available to the internet increases the importance to consider the security of them.

Multiple researchers have already looked into the security of several OPC UA implementations. In 2022, researchers from the German Federal Office for Information Security (BSI), wrote an extensive report about the security of various implementations and applications of the OPC UA protocol, including a section about fuzzing these implementations [7]. Researchers from Kaspersky Lab ICS CERT also tested the security of the OPC UA protocol. The main focus of their paper, published in 2018, was fuzzing the reference server provided by the OPC Foundation [3]. We discuss the works by BSI and Kaspersky Lab in Chapter 3.

Most research that has been done about fuzzing OPC UA is either a few years old, or lacks detail about the fuzzing process. The nature of open source software also means that such software is often frequently updated. To ensure relevancy of this research, it is important to describe the process of testing the security of the several OPC UA implementations. This way, these experiments could be repeated in the future to assess the security of future versions of these implementations. Therefore, in this thesis, the research question **How do we effectively fuzz open source implementations of the OPC UA protocol?** will be answered.

In Chapter 2, required background information for this thesis will be explained. This includes information about the workings of the OPC UA protocol that is required for fuzzing the OPC UA implementations, information about sanitizers and general fuzzing terminology. In Chapter 3, we discuss related work. Specifically, we discuss research which has already been done about the security of OPC UA. In Chapter 4, we perform black-box fuzzing on seven OPC UA implementations using Boofuzz. In Chapter 5, we use AFLNet, a fuzzer only capable of fuzzing C and C++ programs, to perform grey-box fuzzing on three OPC UA implementations written in C and C++. Chapter 6 discusses future work and Chapter 7 contains the conclusions of this thesis.

2 Background

In this chapter, we explain the background information required to understand the experiments in this thesis. In Section 2.1, background about sanitizers is given. In Section 2.2, fuzzing terminology is defined. In Section 2.3, important details about the OPC UA standard that are required for fuzzing OPC UA implementations are explained.

2.1 Sanitizers

Sanitizers are tools that can be specified during compile time that add checks to a program when it is run. Languages such as C and C++ do not check if the developer is doing something it should not do, such as accessing an array out of bounds. While the behavior of these languages improves performance, it also makes it more likely for programmers to make mistakes that can cause bugs. Sanitizers add a set of checks to the code that will cause the program to report various errors that C/C++ compilers do not check for by default.

Sanitizers are supported by both the GCC and Clang compilers. In most cases, they can simply be enabled by adding a flag to the compiler. However, most sanitizers cost a significant amount of performance while using it with a program. An overview of popular sanitizers is given in Sections 2.1.1 to 2.1.4.

2.1.1 AddressSanitizer

AddressSanitizer (ASAN) [24] was introduced in 2012 by researchers at Google. ASAN detects out-of-bounds accesses of memory and use-after-free bugs, the latter of which occurs when a program uses a pointer whose destination that has been freed. If either out-of-bounds access or use-after-free occurs, ASAN crashes the program and reports the problem. According to the researchers from Google, using ASAN causes an average slowdown of 73%.

2.1.2 MemorySanitizer

MemorySanitizer (MSAN) [26] was introduced in 2015, also by a group of researches at Google. MSAN detects uninitialized reads, which occurs when we initialize a variable in C/C++ and read from it without first assigning a value to it. For example, if we initialize an array in C, it will not initialize its values: the values of the array will initially be set to what was in that exact spot in memory before it was initialized. These values are unpredictable and bugs could occur if the developer assumes that the values are, for example, 0. Since the initial values of variables are quite often 0, these bugs do not always occur. If MSAN detects such a read, it will crash the program and print a report stating where the uninitialized read occurred. MSAN causes an average slowdown of 250%.

2.1.3 UndefinedBehaviorSanitizer

UndefinedBehaviorSanitizer (UBSAN) detects various types of undefined behavior in C/C++. Examples of undefined behavior include:

- `bool` values being set to other values than `0` or `1`
- Functions not returning values while a return type has been defined
- Usages of null pointers

UBSAN can be configured such that when it detects undefined behavior, it will print an error and crash the program. The documentation[1] for Clang contains a list of available configuration options. It also contains a complete list of undefined behaviors that UBSAN detects.

2.1.4 ThreadSanitizer

ThreadSanitizer (TSAN) [25] is used for detecting data races in multithreaded C/C++ applications. By default, TSAN will print a report both when it finds a data race. It will not crash the program as soon as it finds a data race. Instead, it prints a full report with all found data races after the program has finished running. If any data races were reported, TSAN changes the exit code of the program so it can be easily detected by other tools.

2.2 Fuzzing

In Sections 2.2.1 to 2.2.5, we determine the fuzzing terminology and techniques that are used throughout this thesis.

2.2.1 General idea

When fuzzing a System Under Test (SUT), we send various malformed inputs to the SUT in hopes of crashing it [6, 17]. These crashes could reveal several types of bugs in the SUT. For example, in memory unsafe languages, a buffer overflow could occur, which could then be used to crash the program and could cause denial of service. In more sophisticated attacks, this kind of bugs could be abused by an attacker to take over the system.

2.2.2 Black-box fuzzing

When we black-box fuzz a SUT, we do not have or use any knowledge about the internal workings of said SUT [28]. A black-box fuzzer provides input to the SUT and can only observe the program from outside. A black-box fuzzer can only read the output the program gives, which includes observing crashes.

2.2.2.1 Grammar-based fuzzing

Black-box fuzzing is often grammar based. The most basic form of fuzzing would be to provide completely random input to the SUT. However, the large number of possible inputs a program can have when fuzzing would make this method of fuzzing unfeasible in most cases. Furthermore, a lot of programs and protocols depend on a specific format, such as XML, and simply reject packets that stray away from that format. With grammar based fuzzing, the expected input format must be determined and provided beforehand, including each of the fields within said format [14]. Depending on the fuzzer, it is also possible to determine a grammar for the state model of the program.

An example of a fuzzer capable of black-box fuzzing is Boofuzz [22], which we used to fuzz an OPC UA implementations in Chapter 4. Boofuzz accepts grammar written with a specific Python syntax. If we would for example want to fuzz the OPC UA message type `OpenSecureChannelRequest`, we would need to provide a grammar for that message and the `Hello` message type preceding it. Furthermore, we would also need to specify to specify the state model. In this case, we should specify in the grammar that `OpenSecureChannelRequest` follows `Hello`.

[1] `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`

2.2.2.2 Dumb mutation-based fuzzing

Mutation-based fuzzing is a form of fuzzing where the fuzzer takes an input and mutates it. We do not provide a grammar of the SUT for this kind of fuzzer. Instead we provide seed inputs: actual inputs that could be given to the SUT. Examples of tools that aid with or perform mutation based fuzzing are Radamsa [15] and zzuf [16]. We could consider these tools as 'dumb' mutation based fuzzers: they do not use feedback from the SUT to adjust their inputs. An example of several mutations generated by Radamsa has been provided in Figure 1.

```
$ echo "The current time is 12:00." | radamsa
The current time is 12:4294967297.
The current time is 12:0.
$ echo "The current time is 12:00." | radamsa
The current time is 12:3182974.
$ echo "The current time is 12:00." | radamsa
The current time is 128:-170141183460469231731687303715884105728.
```

Figure 1: Examples of mutations generated by Radamsa

2.2.3 White-box fuzzing

White box fuzzers commonly use symbolic execution in order to effectively explore different branches in the SUT. Reaching the branch containing the call to `abort` in the program shown in Figure 2 could take a long time when using a traditional black-box fuzzer, as the fuzzer would A white-box fuzzer, would perform symbolic execution in an attempt to reach as many paths as possible within the SUT. As such, a white-box fuzzer would be able to reach the call to `abort` in the example program more quickly than a black-box fuzzer.

```
void function(int a) {
  if (a == 1234) {
    abort();
  }
}
```

Figure 2: Example program

2.2.4 Grey-box fuzzing

Grey-box fuzzers, also known as coverage guided fuzzers, leverage data about the execution of the SUT to attempt to reach as many different parts of the code as possible. AFL [30] is a mutation-based grey-box fuzzer which uses a genetic algorithm to generate test cases, of which it then measures the edge-coverage inside the SUT. An edge is a single 'path' a program can take. For example, in the program shown in Figure 2, the if-case introduces two edges: the edge where `a == 1234` is true leading to the `abort()` call and an edge where it is false, leading to the function returning.

AFL measures edge-coverage by injecting instrumentation into the SUT, which causes the SUT to report each encountered edge back to AFL. AFL then uses this data to compare different mutated inputs and prioritizes the input which discovered the most new edges in order to discover different execution paths in the SUT more effectively.

2.2.5 Stateful fuzzing

In fuzzing terminology, we can distinguish between stateless and stateful fuzzers [6]. Stateless fuzzers assume that past inputs to a SUT do not affect the outcome of future inputs. Stateful fuzzers, on the other hand, aim to explore the state space of a SUT.

Stateless fuzzing is often faster than stateful fuzzing, since the fuzzer does not have to reset the state of the SUT by for example restarting it, as it assumes it is stateless. *Stateful fuzzing*, on the other hand, finds deeper errors in stateful binaries, as those fuzzers attempt to cover the state model as much as possible.

There are several ways for a fuzzer to achieve a high state coverage. *Grammar-based fuzzing*, as performed in Chapter 4, requires a pre-defined definition of the state model. The fuzzer has to know a path from the initial state to the target state we want to fuzz. *Coverage guided fuzzers*, as used in Chapter 5, do not require a pre-defined grammar, but instead use seeds and feedback from the program in order to deduct the state model and explore it.

In the example state model shown in Figure 3, a stateless fuzzer might take path A and get stuck in the loop at s_1. A stateful fuzzer, on the other hand, uses knowledge of the state model to navigate from the starting state to the other states described in the state model. A stateful fuzzer might for example first take path A, fuzz state s_1, then restart the SUT, take path B and fuzz state s_2.

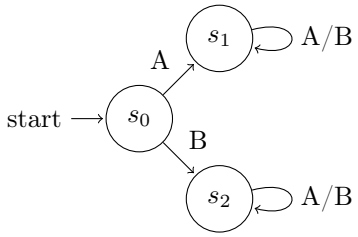

Figure 3: Example state model

An example of a stateful fuzzer is AFLNet [23], which has been used in Chapter 5, and was able to find reproducible crashes in two different OPC UA implementations, as shown in section Section 5.3. Other examples of stateful fuzzers include, SGPFuzzer [29] and SGFuzz [1]. These fuzzers could be used in future work, as described in Section 6.5.

2.3 OPC UA

In this section, important details of the OPC UA standard are explained. In Section 2.3.1, an overview of the OPC UA protocol is given. Section 2.3.3 contains an overview of the different message types that are used within the OPC UA standard. Section 2.3.2 contains an explanation of important service sets defined within the OPC UA standard. Section 2.3.4 contains the state machine of the OPC UA standard.

2.3.1 Overview

Open Platform Communications Unified Architecture (OPC UA) is a standard used for enabling multiple devices in an ICS to communicate with each other. After a connection is facilitated between two or more devices, various types of data and messages can be sent, such as sensor-data or commands to control one or more devices.

2.3.2 Service sets

The OPC UA standard defines several service sets. Each service set is essentially a group of message types that are used to fulfill one purpose within an OPC UA connection. Some examples of commonly used service sets, along with the message types that accompany them, have been listed in Section 2.3.3. Sections 2.3.2.1 to 2.3.2.3 discuss the SecureChannel, Discovery and Session service sets. The usage of these service sets are required to establish a connection between a client and a server. The OPC UA specification[2] specifies a complete list of service sets.

2.3.2.1 SecureChannel Service Set

The SecureChannel messages are sent to either open or close a secure channel between a server and client. These are described in section 5.5 of the OPC UA specification[3]. Secure channels support several message security modes[4]:

- None
- Sign
- Sign and encrypt

Note that a secure channel is set up and used even when security mode *None* is used. This secure channel will not have encryption or message verification enabled. When fuzzing OPC UA implementations, it is important that security mode *None* is used, as this avoids us having to sign and possibly encrypt messages while fuzzing in order to get the OPC UA implementation to accept them.

2.3.2.2 Discovery Service Set

The messages from the Discovery Service Set[5] are used by the client to discover information about the server without having to establish a secure channel beforehand. For example, a client can request a list of endpoints from the server that it could connect to. The client could also request data, such as a list of supported encryption methods, that is required to establish a secure channel. Note that, while a secure channel must be established for this service set, this secure channel will not use encryption.

2.3.2.3 Session Service Set

The Session Service Set[6] is responsible for authenticating users. The service set first creates a session. Then, it activates the session by authenticating the user. Authentication is performed using a UserIdentityToken[7]. Four types of authentication methods are defined in the OPC UA standard:

- Anonymous
- Username and password
- X.509 Certificate
- Issued identity, provided by an external authorization service (e.g. OAuth2)

[2] `https://reference.opcfoundation.org/Core/Part1/v105/docs/7` (Client/Server Service Sets)
[3] `https://reference.opcfoundation.org/Core/Part4/v105/docs/5.5` (SecureChannel Service Set)
[4] `https://reference.opcfoundation.org/Core/Part4/v105/docs/7.20` (MessageSecurityMode)
[5] `https://reference.opcfoundation.org/Core/Part4/v105/docs/5.4` (Discovery Service Set)
[6] `https://reference.opcfoundation.org/Core/Part4/v105/docs/5.6` (Session Service Set)
[7] `https://reference.opcfoundation.org/Core/Part4/v105/docs/7.41` (UserIdentityToken parameters)

2.3.3 Message types

The OPC UA reference defines multiple basic message types (Hello, Acknowledge, Error, ReverseHello) that can be used within a message header[8].

In order to limit the scope of this research, we mostly limit ourselves to the following 4 sets for a total of 9 messages. These messages are all related to establishing connections with an OPC UA server. This is the same set of messages that BSI used [7] for black-box fuzzing as discussed in Section 3.1. Fuzzing more message types and service sets has been left as future work, which is discussed in Section 6.2.

- *Basic message types*
  1. Hello
- *SecureChannel service set*
  2. OpenSecureChannel
  3. CloseSecureChannel
- *Discovery service set*
  4. FindServers
  5. FindServersOnNetwork
  6. RegisterServer2
  7. GetEndpoints
- *Session service set*
  8. CreateSession
  9. ActivateSession

2.3.4 State machine

OPC UA is a stateful protocol. Multiple 'layers' of connections have to be initiated before a client and server can communicate properly. Firstly, the client has to request an unencrypted connection using the Hello message. It then has to request and open a secure channel for further communication with the server. At this point, the client can perform actions from the discovery service set as discussed in Section 2.3.2.2. The client can also choose to authenticate and open an authenticated session with the server, after which it can interact with the OPC UA application it has connected to.

Using section 7.1.3 from the OPC UA specification[9] and a network capture of an OPC UA client and server, we reconstructed the state machine from the perspective of the client. This state machine can be seen in Figure 4.

[8] `https://reference.opcfoundation.org/Core/Part6/v105/docs/7.1.2.2` (Message Header)
[9] `https://reference.opcfoundation.org/Core/Part6/v105/docs/7.1.3` (Establishing a connection)

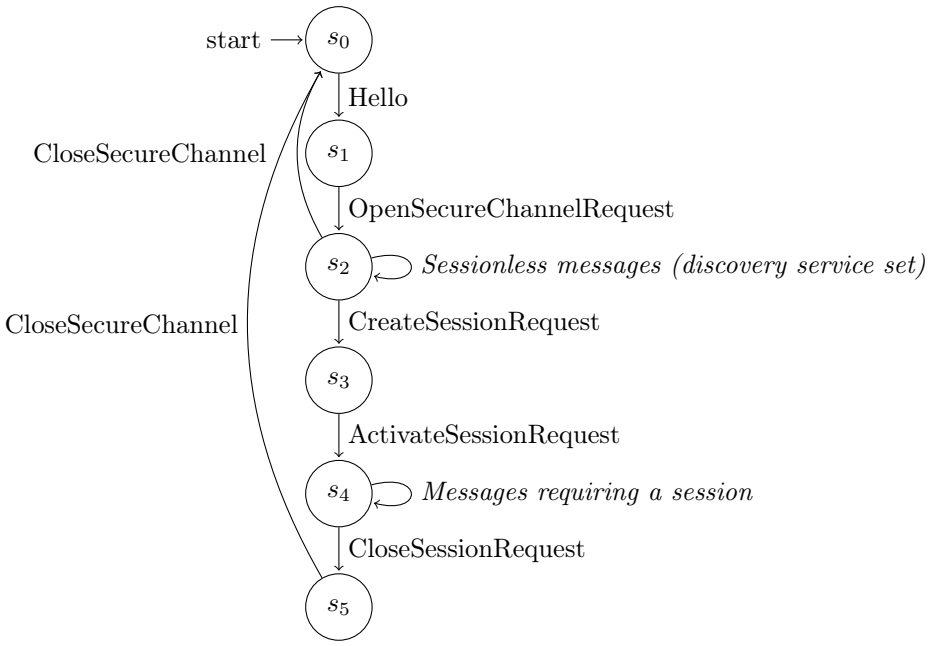

Figure 4: Manual reconstruction of the OPC UA state machine

3 Related Work

In this chapter, we discuss related work on analyzing the security of several OPC UA implementations. Section discusses two security analyses on several aspects of OPC UA. Section Section 3.2 discusses a fuzzing campaign performed by Kaspersky Lab ICS CERT. Section Section 3.3 discusses work by Mühlbauer et al. which analyzed four OPC UA implementations based on specific aspects. Section 3.4 gives an overview of the OPC UA implementations used in this thesis and in the related work. In Section 3.5, we compare these works and list their contradictions and similarities.

3.1 German Federal Office for Information Security

In 2017 the German Federal Office for Information Security [5], also known as Bundesamt für Sicherheit in der Informationstechnik (BSI) in German, published an extensive security analysis of OPC UA. They analyzed both the OPC UA specification and the official OPC UA ANSI C implementation.

- BSI analyzed the OPC UA specification by analyzing how the OPC UA specification covers specific security features and checking the OPC UA specification for errors and unclear descriptions. They concluded that their analysis revealed no systematic errors in the specification. However, several improvements were recommended. For example, BSI suggested to add a note to the specification about outdated and insecure algorithms such as SHA1.
- The official OPC UA ANSI C implementation was examined by first testing if the implementation properly accepts and rejects different certificates. It was found that the C implementation accepted various types of incorrect certificates as correct.
- BSI also performed both static code analysis and found that the implementation used a deprecated function that could cause issues with multi-threading, if used.
- BSI fuzzed the implementation using Peach [21]. The fuzzer found that the ANSI C implementation had several deviations from to the OPC UA specification. Two of those were security related. One of the found issues was that the server accepted messages with an incorrect sequence number.
- Finally, BSI re-performed the fuzzing experiments with Valgrind. Several issues were found. According to BSI, these could cause crashes. However, they stated that more evaluation is required in order to determine the impact the found issues could cause.

In 2022, BSI re-performed their OPC UA security analysis.

- Most changes suggested in their initial research have been implemented in the specification. BSI looked at the OPC UA specification again, focusing on security related changes since the last analysis. Multiple disparities, unclear or lacking descriptions were found within the same specification. Most significantly, BSI found that the specification lacks an overview of all security features in one place. Instead, information about security features is spread between 8 different chapters in the OPC UA specification.
- BSI performed a black-box fuzzing campaign on five open source OPC UA implementations. They examined the official .NET Standard Stack, open62541, Eclipse

Milo, python-opcua and node-opcua. They used Boofuzz for this and have released the accompanying setup on GitHub[10]. They specifically fuzzed 9 message types, which have also been listed in Section 2.3.3. Many crashes were found that were not reproducible outside the testing environment. Notably, they did not use sanitizers while fuzzing. Some reproducible crashes were found for node-opcua, which were solved by the authors of the implementation after BSI reported the bugs to them.[11]

- BSI also performed a grey-box fuzzing campaign, which they mistakenly called white-box fuzzing in their report. They used libFuzzer with ASAN to fuzz the open62541 OPC UA implementation written in C. Several crashes were found, including a heap buffer overflow.

3.2 Kaspersky Lab ICS CERT

In 2018, researchers from Kaspersky Lab ICS CERT [3] also ran a fuzzing campaign on the reference OPC UA ANSI C implementation. They fuzzed the legacy OPC UA ANSI C implementation, on top of sample applications provided by the OPC Foundation, using AFL. To make this work, they overloaded the networking functions to read data from local files instead, which also increased the fuzzing speed. They also compiled the OPC UA ANSI C implementation using AddressSanitizer. As a result, nine vulnerabilities were found in the tested OPC UA applications. They also tested several commercial products and found that some of these products borrowed code from examples provided by the OPC Foundation found similar security issues.

In 2020, Kaspersky Lab [2] described how they fuzzed OPC UA applications. They fuzzed an example server on the legacy OPC UA ANSI C implementation by specifically fuzzing the data handling functions in that implementation. For their report, they used libfuzzer on Windows to perform the testing. After setting up the fuzzer, they managed to find a crash within a few minutes. Like BSI, the code they used has been provided on GitHub[12]. Note that the ANSI C implementation they tested is no longer developed by the OPC Foundation and as such, is no longer available for download. Nonetheless, their research could be applied to other C/C++ OPC UA implementations.

3.3 Mühlbauer et al.

Mühlbauer et al. [18] analyzed the security of four open source OPC UA implementations, as listed in Section 3.4. They inspected several aspects of these implementations. They mainly analyzed the dependencies of the tested implementations, timeouts, available security policies, how messages are processed and randomness used within the implementations.

The analysis of how messages are processed is the most relevant for us. The researchers used the source code of the examined OPC UA implementations to check if messages are processed according to the OPC UA standard. For example, they checked if incorrect message types make the server return an error, instead of silently being accepted by the server.

A few shortcomings were found. Mainly, they found that node-opcua, an OPC UA implementation written in JavaScript, had less strict packet checks than specified in the OPC UA standard. Specifically, a decryption function within the OPC UA implementation ignored the incoming message type, always assuming it was a secure channel message that should be decrypted. However, they concluded that at the time their

[10] `https://github.com/fkie-cad/blackbox-opcua-fuzzing`
[11] `https://github.com/node-opcua/node-opcua/releases/tag/v2.47.0`
[12] `https://github.com/KL-ICS-CERT/opcua-fuzzing-example`

research was published, the open source implementations they tested were sufficiently secure.

3.4 Overview of OPC UA implementations

Various open source implementations for the OPC UA standard exist. An extensive list of OPC UA implementations has been documented by Erba et al. [8]. In Figure 5, we list various open source OPC UA implementations that we encountered in the related work and implementations which we used in this thesis.

		Used in					
Implementation	**Language**	Thesis	BSI 2017 [5]	BSI 2022 [7]	Kaspersky 2017[3]	Kaspersky 2020[2]	Mühlbauer et al. [18]
OPC UA .NET Standard Stack [11]	C#	*		*			*
open62541 [20]	C	*		*			*
Eclipse Milo [9]	Java	*		*			
python-opcua [12]	Python	*		*			*
node-opcua [19]	JavaScript	*		*			*
FreeOpcUa [13]	C++	*					
legacy OPC UA ANSI-C Stack [10]	C	*	*		*	*	

Figure 5: Table of OPC UA implementations and their occurrences within related works

Besides the .NET Standard Stack, the OPC Foundation also used to provide OPC UA implementations in other languages. The legacy OPC UA ANSI-C Stack [10] is commonly discussed in older related works. However, this specific implementation is no longer maintained by the OPC Foundation and has been discontinued since 2021 in favor of the .NET Standard Stack and other open source implementations. However, a slimmed down version of the legacy OPC UA ANSI-C Stack is still used in the actively maintained OPC Foundation LDS (Local Discovery Server) Stack[13]. Furthermore, it is not unlikely that other implementations and software exists which uses the legacy OPC UA ANSI-C Stack as either a dependency or as a basis.

3.5 Discussion

Interestingly, in the work by Mühlbauer et al. from 2020, potential issues are found in node-opcua, while in 2022, BSI found several crashes in the same implementation. Mühlbauer et al. found that node-opcua had less strict packet checks than specified in the OPC UA standard. BSI later found crashes using black-box fuzzing. Upon inspection of the changes made in the node-opcua version that fixed the crashes found by BSI[14], it appears that the shortcomings found by Mühlbauer at al. are not related to the crashed found by BSI. Version v2.47.0 of node-opcua specifically fixed multiple untrapped exceptions of various message types that are sent outside a secure channel, such as the Hello message.

[13] `https://github.com/OPCFoundation/UA-LDS`
[14] `https://github.com/node-opcua/node-opcua/releases/tag/v2.47.0`

4 Black-box fuzzing OPC UA using Boofuzz

Re-doing the black-box fuzzing experiments from BSI allows us to determine if the results of the tested OPC UA implementations have improved. Due to the relatively small time-frame between the experiments from BSI and this thesis, we do not expect large changes in the results. However, even if no errors are found, it will still allow us to set a baseline for further experiments. Secondly, we tested two more implementations, FreeOpcUA and the legacy OPC UA ANSI-C stack, the latter of which has not been updated since 2018 and should have a higher chance of Boofuzz finding crashes in it. Thirdly, we run Boofuzz on open62541 (C) with sanitizers, which might reveal more errors. However, the chance of finding bugs in open62541 is low since this project is continuously fuzzed by Google's OSS-Fuzz[15].

The reason we have chosen to redo and extend upon the black-box fuzzing experiments by BSI is that they have provided all code used to setup Boofuzz on GitHub[16], making it easier to reproduce then when we have to rely on just the paper. Furthermore, the research BSI performed is general in the sense that the used fuzzer should be compatible with most OPC UA implementations. Comparatively, Kaspersky Labs as discussed in Section 3.2, did not provide their setup outside of their paper and also exclusively focused on the outdated legacy OPC UA ANSI-C stack, thus making it less interesting to repeat.

In this chapter, we redo the black-box fuzzing experiments performed by BSI [7] in 2022, which we discussed in Section 3.1. In Section 4.1, we discuss the software and hardware used on our machines to perform the experiments. In Section 4.2, we discuss how the GitHub setup made for BSI works and can be run. In Section 4.4 we re-run the experiment using the open62541 (C) OPC UA implementations using various sanitizers. We discuss the results in Section 4.5, and give our conclusions about the Boofuzz experiments in Section 4.6.

4.1 System setup

Each of the experiments in this chapter has been performed on a virtual machine. Using such an isolated environment is generally more secure than running it on the host computer, as crashes of the SUT will not impact the operating system of the host and should in the worst case only be able to crash the virtual machine. Furthermore, the usage of a virtual machine reduces the number of variables that interfere with the fuzzing process. However, the performance of virtual machines is not entirely independent of the host machine. For example, if the host machine is running a CPU intensive task, the host's operating system might prioritize this task over the task running in the virtual machine itself. As a result of this, fluctuations of performance can occur in the virtual machine dependent on the host machine, which might cause inaccuracies in the measured times.

We used VMWare Workstation Player to run the virtual machine for these experiments. The virtual machine used for fuzzing the OPC UA implementations using Boofuzz was running Ubuntu Linux 22.10. The virtual machine was assigned 8 CPU-cores and 8 GB of RAM.

[15] `https://github.com/google/oss-fuzz`
[16] `https://github.com/fkie-cad/blackbox-opcua-fuzzing`

4.2 Boofuzz setup

For performing the Boofuzz experiments within the virtual machine, we are using the same scripts and Docker environment that was used in the report by BSI [7]. As discussed in Section 3.1, this code has been open sourced on GitHub[17] by Fraunhofer FKIE, who performed the black-box fuzzing for the report written by BSI. A flowchart of this setup has been given in Figure 6.

The state-grammar for Boofuzz provided by BSI covers the following states from the OPC UA state machine given in Figure 4:

1. Hello
2. Hello → OpenSecureChannelRequest
3. Hello → OpenSecureChannelRequest → CloseSecureChannel
4. Hello → OpenSecureChannelRequest → FindEndpoints *(Sessionless message)*
5. Hello → OpenSecureChannelRequest → GetEndpoints *(Sessionless message)*
6. Hello → OpenSecureChannelRequest → FindServersOnNetwork *(Sessionless message)*
7. Hello → OpenSecureChannelRequest → RegisterServer2 *(Sessionless message)*
8. Hello → OpenSecureChannelRequest → CreateSessionRequest
9. Hello → OpenSecureChannelRequest → CreateSessionRequest → ActivateSessionRequest

For each message sequence given above, only the last message is fuzzed by Boofuzz. The other messages are used to get to that state. With this state grammar, most states from Figure 4 are covered. The CloseSessionRequest transition and following state s_5 are not covered, however. Boofuzz also does not cover messages requiring a session.

We run the provided Boofuzz setup to fuzz the *open62541* implementation and keep track of the time, using the following command:

```
$ time python3 run_docker_fuzzing.py --path ./results/open62541 open62541
```

After running this command, a Docker container will be created, in which the provided implementation will be installed. After the implementation and the accompanying OPC UA example server are built automatically. The OPC UA example server will be launched using the Boofuzz process monitor, which watches the process for any crashes. Simultaneously, the provided Boofuzz script, which contains all required grammar, message sequences and fuzzing parameters will be executed.

The Boofuzz script will then try to perform 32.389 test cases, where each test case consists of a message sequence to be sent to the SUT. This reliance on pre-defined sequences means that Boofuzz does not fully explore the state space of the SUT, unlike Grey- and White-box fuzzers described in Section 2.2 do. For each test case, Boofuzz will either wait for a response for the server, or retrieve a signal from the process monitor that the process has crashed.

After sending the message, Boofuzz will wait for a response from the SUT. If it succeeds, it will simply move on to the next test case. If a crash of the SUT is detected by the process monitor, or if no response by the SUT is detected within 5 seconds, the test case will be considered as a failure by Boofuzz. On failure, the SUT will be killed, after which the message sequence will be logged. Furthermore, the script will attempt to replay the crash on a newly started server and stores the result.

After all tests cases have run, the results will be exported to a SQLite database. Additionally, a list of failed test cases, including the sent sequence of messages, reason

[17] `https://github.com/fkie-cad/blackbox-opcua-fuzzing`

for failure and if the crash was reproducible will be stored in a separate file in JSON format.

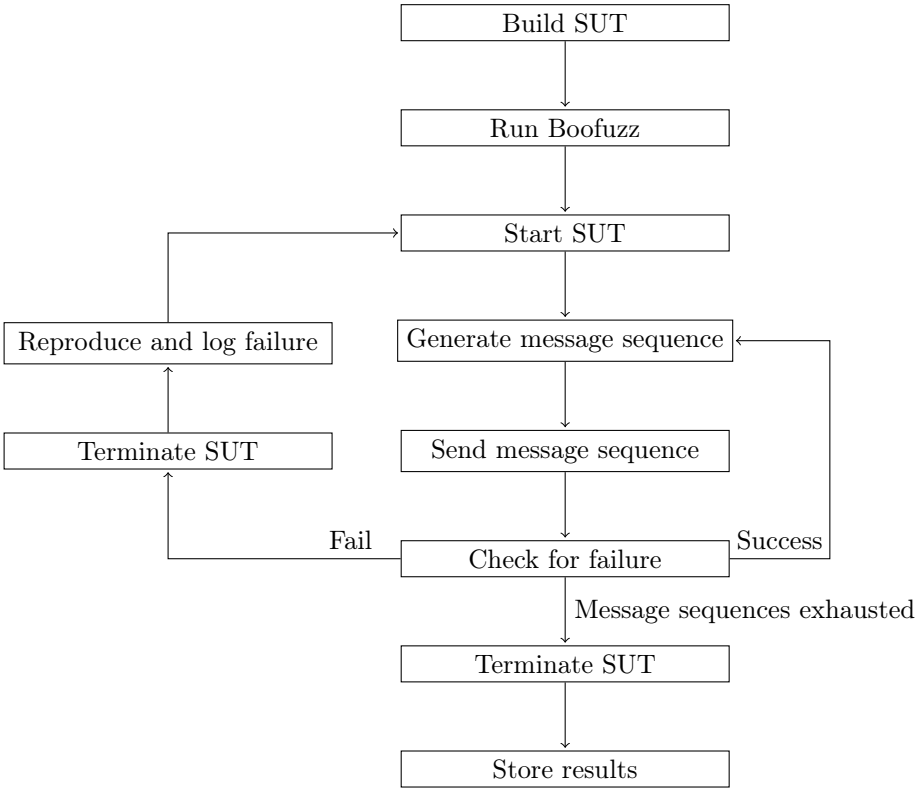

Figure 6: Flowchart of Boofuzz setup described in Section 4.2.

4.3 Adding support for new implementations

The OPC UA implementations used as SUTs by Boofuzz each have their own subfolder with installation files within the `targets` folder of the setup. For each OPC UA implementation, this folder contains an installation script named `install.sh` and can hold other files required to build said OPC UA implementations as well. When starting Boofuzz using `run_docker_fuzzing.py` as described in 4.2, the files from the `targets` folder are copied and the `install.sh` file is run.

The structure of the Boofuzz setup provided by BSI allows us to easily add new implementations. This can be done by performing a few steps:

1. Create a folder in `targets` with the name of the implementation.
2. Create an installation script named `install.sh`. This installation script usually does the following things:
   (a) Use the APT package-manager to install dependencies required for building and running the OPC UA implementation.
   (b) Download the source code of the OPC UA implementation by cloning a GitHub repository.
   (c) Build the OPC UA implementation.

(d) Copy the server binary of the OPC UA to `/opt/app/target/` so Boofuzz can find it.

3. Add the name of the OPC UA implementation to `run_docker_fuzzing.py` so that this script knows where the find the installation files.

Using this method, we added support for fuzzing the OPC UA implementations FreeOpcUa and the legacy OPC UA ANSI-C Stack using BSI's Boofuzz setup. The installation scripts we used can be found in Appendix B. We also used this method to add separate targets that compile open62541, FreeOpcUa and the legacy ANSI-C OPC UA Stack using the sanitizers ASAN and UBSAN in Section 4.4.

4.4 Compiling OPC UA implementations using sanitizers

As discussed in Section 3.1, BSI did not use sanitizers during their black-box fuzzing experiments. Sanitizers add additional run-time checks to the program it is compiled with, as discussed in Section 2.1. Compiling the SUT using sanitizers thus increases the chance of finding bugs through fuzzing. Note that the sanitizers we discussed in Section 2.1 can only be added to C/C++ implementations.

We compiled open62541, FreeOpcUa and the legacy OPC UA ANSI-C Stack with ASAN and UBSAN separately. For each C/C++ implementation, we added two additional 'target' folders with accompanying installation scripts to the Boofuzz setup as described in Section 4.3. For open62451, we did this by adding a parameter to `cmake` command used to build the implementation. For example, to add ASAN to open62541, we would add the following C-flags to the `cmake` command in the implementation's installation script:

```
cmake -DCMAKE_C_FLAGS="-fsanitize=address -fno-omit-frame-pointer" -
    DBUILD_SHARED_LIBS=ON -DUA_BUILD_EXAMPLES=ON ../open62541
```

Similarly, to add ASAN to both FreeOpcUa and the legacy OPC UA ANSI-C Stack, we would change the `cmake` command in the implementation's installation script to either of the following:

```
# C implementations (legacy OPC UA ANSI-C Stack):
cmake -DCMAKE_C_FLAGS="-fsanitize=address -fno-omit-frame-pointer" ..

# C++ implementations (FreeOpcUa):
cmake -DCMAKE_CXX_FLAGS="-fsanitize=address -fno-omit-frame-pointer" ..
```

To use UBSAN, we use the C/C++-flag `-fsanitize=undefined` instead. Due to time-limitations, we did not repeat the experiments using other sanitizers.

4.5 Fuzzing results

Implementation	Version	Language	Failures		Runtime
			Total	Reproducible	
.NET Standard Stack	7550c65	C#	⁄	⁄	⁄
open62541	46d0395	C	95	0	09:49:01
open62541 (ASAN)	46d0395	C	94	0	09:50:29
open62541 (UBSAN)	46d0395	C	94	0	09:49:43
legacy OPC UA ANSI-C Stack	1.04.342	C	94	0	08:48:46
legacy OPC UA ANSI-C Stack (ASAN)	1.04.342	C	91	0	08:48:55
legacy OPC UA ANSI-C Stack (UBSAN)	1.04.342	C	90	0	08:57:11
FreeOpcUa	6eac097	C++	73	0	08:39:38
FreeOpcUa (ASAN)	6eac097	C++	72	0	09:33:38
FreeOpcUa (UBSAN)	6eac097	C++	66	0	09:32:00
Eclipse Milo	0.5.0	Java	85	0	19:06:07
python-opcua	fdf5f3c	Python	83	0	21:05:27
node-opcua	v2.46.0	JavaScript	18	18	05:22:16
node-opcua	v2.47.0	JavaScript	0	0	14:01:14
node-opcua	v2.107.0	JavaScript	12	0	08:44:29

Figure 7: Black-box fuzzing results with Boofuzz

As can be seen in the table in Figure 7, Boofuzz could only find reproducible failures in an old version of node-opcua. Note that we call these failures and not crashes. Boofuzz counts a result as a failure when the SUT crashes, when the SUT responds with empty messages or when the SUT does not respond to messages at all. Most of the failures that were found by Boofuzz are related to connection and session limits set by the implementations. Due to the nature of fuzzing, many sessions are not closed properly and would be left open on the server side. This is especially likely to happen when fuzzing messages that are supposed to close a session, as many of these messages should be rejected by the server for being malformed, thus keeping the session open indefinitely. When these limits are hit, the implementation might not respond to further session requests, which Boofuzz counts as a failure.

Furthermore, we could not get the .NET Standard Stack to run within the Boofuzz setup. After attempting to build the .NET Standard Stack, it states that some dependencies are missing. We decided not to look into this any further.

Boofuzz was able to find several failures within node-opcua. We tested 3 versions of node-opcua: the version where the bugs reported by BSI were fixed, a version before that and a newer version. Within the BSI report, 13 reproducible failures were found for node-opcua. We found 18 reproducible failures. This difference could potentially be explained by a mismatch in tested versions between the BSI report and this thesis, as BSI did not mention the specific version of node-opcua they used for testing, while we tested the version before the issues were resolved. Since no failures could be found in the next version, no further investigation into this is needed.

4.6 Conclusions

In the introduction of this experiment, we mentioned several goals we aimed to achieve. Firstly, we aimed to reproduce the results by BSI on more recent versions of the implementations they tested. We were only able to reproduce the failures on an old version of node-opcua that BSI had also found in their research, but were not able to find new failures in newer versions of the OPC UA implementations that they tested. Secondly, we tested two additional implementations, namely the legacy OPC UA ANSI-C stack and FreeOpcUa. No reproducible failures were found in these implementations. Thirdly, we repeated the tests of open62541 with two different sanitizers, which also did not yield any additional failures.

Setup. As described in Section 4.2, we used the Boofuzz setup used in BSI's research as a basis for this experiment. This setup provided by BSI uses Docker containers, which makes it simple to reset the container if something went wrong with running or modifying the setup. Furthermore, if configured properly, it ensures that the experiments can be performed consistently and independent of the host operating system. This is especially useful, as different OPC UA implementations require different versions of dependencies which might conflict with one another otherwise. One disadvantage of Docker containers is that it was difficult to get access to the web interface of Boofuzz in order to view the fuzzer's progress. To get around this, we ran `curl` in the container to fetch the web-page and view this progress.

Reproducibility. Re-running the experiments performed by BSI could be done by running a single Python-script which should automatically sets up a Docker container with the chosen OPC UA implementation and runs Boofuzz in it. The work by BSI was not fully reproducible using the setup from their GitHub repository. The installation script of some implementations, such as node-opcua, were not hardcoded to install a specific version of the OPC UA implementation. node-opcua's installation script had a hardcoded version for the stack, but always downloaded the latest version of the example server, resulting in a version mismatch which initially prevented node-opcua from running within the Boofuzz setup. The .NET Standard Stack was not able to run at all due to a dependency error.

Many non-reproducible failures were found on the several OPC UA implementations we tested using Boofuzz. Many of these non-reproducible failures can be considered normal behaviour. For example, a commonly encountered issue is that the tested OPC UA implementation has a limit on the number of open sessions. Since sessions are rarely properly closed by the fuzzer, these limits are often reached, causing the SUT to refuse new connections, or to simply not respond to them.

Extensibility. The setup from BSI currently supports 9 message types as described in section Section 2.3.3, which are mainly related to server discovery and creating sessions. If we want Boofuzz to fuzz more types of messages, such as messages of different service sets as described in section Section 2.3.2, we would need to define a literal grammar of the structures of this messages. Furthermore, defining a sequence of messages to get to a specific state can be done with a single line of code.

Support for other OPC UA implementations could be added by writing an installation script for the implementation and modifying the Python-script to use the installation script. In most cases, an installation-script script only downloads and builds the chosen OPC UA implementation. For the legacy OPC UA ANSI-C Stack, small modifications were required the source code of the implementation as well to ensure it could be fuzzed properly. This includes disabling the startup-prompt that requires user

interaction in the OPC UA ANSI-C stack. For open62541, we had to specifically disable encryption during the compilation process, as it was enabled by default.

Performance. While running the experiments, we found that the OPC UA implementations open62541, Eclipse Milo, python-opcua and node-opcua performed badly with Boofuzz. For open62541, 32.389 test cases were completed in about 10 hours, averaging 1.11 executions per second. Eclipse Milo and python-opcua took about 20 hours to complete all test cases, while node-opcua was inconsistent between different versions, with runtimes varying between 5 and 14 hours. We found that the legacy OPC UA ANSI-C Stack performed much better, finishing in slightly more than an hour with about 8 executions per second. Interestingly, open6251, Eclipse Milo, python-opcua and node-opcua, had many test cases that had an execution time of exactly 10 seconds.

The execution times of exactly 10 seconds can be attributed to the fact that Boofuzz, by default, has a 5 second timeout for both sending and receiving messages from the SUT. Many of the messages Boofuzz sends are malformed in such a way that the tested OPC UA implementation does not respond to them. This would cause Boofuzz to wait for both timeouts, thus explaining the 10 second execution time for a lot of the test cases. It is likely that these timeouts can be reduced. Determining the right timeout values has been left as future work as described in Section 6.1.

We expected that the usage of a sanitizer such as ASAN would slow down the performance of the fuzzer. However, we found that the time the fuzzer takes to go trough all test cases does not noticeably change when a sanitizer is used. This can also be attributed to the 10 second timeouts, as these timeouts result in an execution time of 10 seconds despite how long the SUT actually needs outside of the time Boofuzz spends waiting.

Interestingly, BSI reported similarly long runtimes, ranging from about 1 to 11 hours. However, they did not consider these runtimes good or bad and did not comment on the potential reasons for these long runtimes.

5 Grey-box fuzzing OPC UA using AFLNet

In this chapter, we perform grey-box fuzzing on the open62541 OPC UA implementation. We fuzz open62541 using AFLNet [23], a fork of AFL [30] with additions for fuzzing network based protocols. As discussed in background Section 2.2.2, coverage guided fuzzers use feedback from instrumentation compiled into the SUT in order to guide which messages will be sent next. Furthermore, AFLNet is a stateful fuzzer and attempts to automatically explore and determine the state space of the program, which, as described in Section 2.2.5, should result in the fuzzer being able to find crashes that occur deeper into the state model. Hence, we expect this fuzzing method to be able to find crashes more effectively than Boofuzz used in Chapter 4, which depends on a predefined grammar for both the messages and state model of the SUT.

In Section 5.1, we discuss how to modify AFLNet in order to perform fuzzing on OPC UA implementations. Section 5.2 discusses how we compiled AFLNet and open62541, and set up and ran both in order to perform grey-box fuzzing. In Section 5.3, we discuss and analyze the found results. In Section 5.4 we analyze the found crashes, and in Section 5.5 we give our conclusions about these experiments.

5.1 Modifying AFLNet

AFLNet has support for several popular protocols, including HTTP, FTP and TLS. However, AFLNet does not support OPC UA by default. Therefore, we need to modify the source code of AFLNet to add support for OPC UA. The readme file on the GitHub of AFLNet[18] describes the steps required to both implement a protocol and to fuzz an implementation of implemented protocols.

In order to modify the source code of AFLNet to implement support for fuzzing OPC UA, we must implement the `extract_requests_opcua` and `extract_response_codes_opcua` functions in both `aflnet.c` and `aflnet.h`. Finally, we must modify the code that handles the protocol parameter `-P` to accept `opcua` as an argument.

The `extract_requests_opcua` function is responsible for breaking up both the provided seed inputs and incoming requests from the SUT into separate messages in order to extract the response codes. For OPC UA, we do this by looking at the length parameter set in the message header, whose structure is shown in Appendix A.2.1. The source code for this function can be found in Appendix D.1.

The `extract_response_codes_opcua` function extracts response codes from OPC UA messages returned by the server. Possible response codes are listed in Appendix A.2 of the OPC UA specification[19]. Since there are more than 2^8 response codes and AFLnet expects an 8 bit integer value, we chose to only return specific response codes for errors, making all successful messages return 0. The source code for this function can be found in Appendix D.2.

[18] `https://github.com/aflnet/aflnet/blob/c2714e9/README.md`
[19] `https://reference.opcfoundation.org/Core/Part6/v105/docs/A.2`

5.2 Installing and running AFLNet

After modifying the source code of AFLNet, we have to perform several steps in order to prepare our SUT for fuzzing. These steps are described in the readme file on the GitHub of AFLNet[20].

In summary, they are:

- Compile and install AFLNet (Section 5.2.1)
- Setup the example client and server of the SUT (Section 5.2.2)
- Take a packet capture between the example client and server and extract the relevant seeds to use as inputs for AFLNet (Section 5.2.3)
- Run AFLNet (Section 5.2.4)

5.2.1 Compiling and installing AFLNet

We install AFLNet using the instructions listed in the documentation in the AFLNet GitHub Repository. Initially, we used the same virtual machine as we used for the Boofuzz experiments as described in Section 4.1. Ubuntu 22.10, which ran in this virtual machine, was not compatible with the build process of AFLNet resulted in various compilation errors, as was also discussed in a GitHub issue for AFLNet[21].

After re-trying the build process on Ubuntu 20.04 and installing the correct dependencies, we built AFLNet successfully. Appendix E contains the script we used to install AFLNet.

5.2.2 Setting up the implementations for fuzzing

We set up three C and C++ implementations for fuzzing with AFLNet. Namely, open62541, the legacy OPC UA ANSI-C Stack and FreeOpcUa. For each of these implementations, we need to ensure that we do the following things:

1. Disable encryption and enable anonymous authorization.
2. Disable session limits and timeouts.
3. Enable them to run on different ports for parallel fuzzing. This can be done by adding a port parameter to the binary, or by creating multiple copies of the binary with different ports.
4. Build the implementations using instrumentation.

For the legacy OPC UA ANSI-C Stack and FreeOpcUa, minimal modifications were required. In order to achieve different ports, we modified the string defining this port in the binary and made different copies with different ports, as this was faster to do than adding a parameter doing this. Do note that for system with more than 8 cores, adding a parameter or automating this method is preferable.

For open62541, we had to perform more modifications. These are described in Section 5.2.2.1.

[20] `https://github.com/aflnet/aflnet/blob/c2714e9/README.md`
[21] `https://github.com/aflnet/aflnet/issues/81`

5.2.2.1 Setting up open62541

The source code of the C OPC UA implementation open62541[22] contains various client and server examples. These examples vary from simple servers and clients that only perform connection and authentication, to more complicated examples where actual data is sent between the client and server in various ways.

We chose to use the `server_ctt` example, where CTT stands for Compliance Testing Tools. This example server was also used in the Boofuzz experiment performed in Chapter 4. The built server binary will be used both for packet captures and for fuzzing. We also need a client so that we can capture communication between the client and the server to use as seed inputs for AFLNet. For this, we will use the `client` example provided by open62541. This specific example connects to the server multiple times and attempts to read and write data to it.

In order to make `server_ctt` more suitable for fuzzing, we changed some configuration settings in the source code of this example server. We specifically remove the limits set by the settings `maxSecureChannels`, `maxSession`. Furthermore, we also removed the 5 second delay the server waits for when shutting it down by removing the `shutdownDelay` configuration option.

We also added a `-port` parameter to `server_ctt`. This parameter enables us to run multiple instances of the server at the same time, each using a different port. We can use this to perform parallel fuzzing with AFLNet, where we run each instance of `server_ctt` with a separate port.

Similarly to the Boofuzz experiment from Chapter 4, we disabled encryption by setting the `DUA_ENABLE_ENCRYPTION` cmake flag to off. However, this caused the example client to refuse connecting to the server stating that it cannot find a suitable UserTokenPolicy for the server. This was caused by the default configurations of the example clients and servers, which expect encrypted authentication even if encryption is disabled. To solve this, we manually enabled anonymous authentication in both `server_ctt` and `client`.

- `server_ctt` should allow anonymous authentication when it is launched using the `-enableAnonymous` flag. Due to a bug, the first parameter is hardcoded such that it is always read as a path for a certificate file. By patching out the code that reads the filename, the anonymous authentication flag can be used successfully.
- To enable anonymous authentication in the client, we have to replace the line containing the call to `UA_Client_connectUsername` with `UA_Client_connect`.

Finally, we built open62541 and instrumented the binaries for fuzzing with AFL using the shell script shown in Appendix C.1.

5.2.3 Generating seed inputs for AFLNet

AFLnet requires seed inputs of communication between a client and server as a starting point. In order to generate these seed inputs, we will capture and dump communication between the client and server of open62541 using tcpdump. We let tcpdump listen on the loopback interface for the server port used by open62541 using the following command:

```
sudo tcpdump -w opcua.pcap -i lo port 4840
```

We then run the modified open62541 example client and server using the following commands in separate terminals:

```
./server_ctt --enableAnonymous
./client_anonymous
```

[22] `https://github.com/open62541/open62541/tree/defd286`

After the client has finished running, we can stop tcpdump, which will make it output `opcua.pcap`. We can then open this file in Wireshark which shows us the captured OPC UA messages and allows us to inspect the individual fields in each of these messages, as shown in Figure 8. By following the TCP stream of these packets in Wireshark, we can see view a full stream from connection to disconnection. With our modified client and server, we see that four distinct TCP streams were captured. The first three TCP streams are related to OPC UA server discovery as discussed in Section 2.3.2.2. The last TCP stream contains actual data sent between the client and the server.

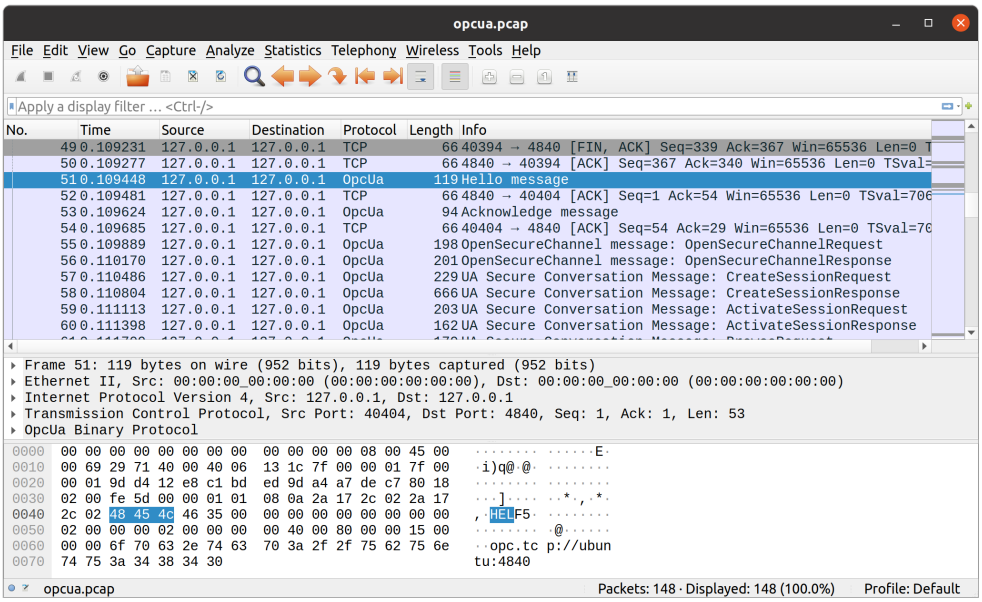

Figure 8: OPC UA communication in Wireshark

In order to extract these streams, we use the 'Follow TCP Stream' option in Wireshark. In the window that opens we filter for packets sent from the client which are the packets with destination port 4840. We then set the display mode to 'Raw' and save the result to a file. This file contains the exact data that has been sent between the client and the server for the specified stream. We repeat this process for all OPC UA streams, extracting all of them to the same folder.

The 3 seed inputs we used covered the following states from the OPC UA state machine given in Figure 4:

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

The third captured trace sends various messages that require a session. The messages that are sent with in this trace that require a session are from the NodeManagement Service Set, Call Service Set and Subscription Service Set. See Section 2.3.2. See 2.3 for background on the service sets defined in the OPC UA standard.

5.2.4 Running AFLNet

Now that we have completed all prerequisites, it is time to run AFLNet. We will be running AFLNet in parallel. On our 8-core machine, this resulted in a performance increase of up to 8 times. AFLNet requires a main instance to run using `-M`, alongside sub-instances using `-S`. We used the following commands to start the main instance of AFLNet:

```
$ afl-fuzz -i in -o out/<name>_sync -M fuzzer0 -N tcp://127.0.0.1/4840 -P OPCUA
    -q 3 -s 3 -E -K -R <binary using port 4840>
```

And we ran the following command in 7 separate terminals to run the sub-instances, where X should be replaced with the numbers 1-7:

```
$ afl-fuzz -i in -o out/<name>_sync -S fuzzerX -N tcp://127.0.0.1/484X -P OPCUA
    -q 3 -s 3 -E -K -R <binary using port 484X>
```

Initially, running AFLNet refused to run due to a misconfiguration related to core dump notifications. Fortunately, AFLNet provides immediate instructions on how to resolve it:

```
[-] Hmm, your system is configured to send core dump notifications to an
    external utility. This will cause issues: there will be an extended delay
    between stumbling upon a crash and having this information relayed to the
    fuzzer via the standard waitpid() API.

    To avoid having crashes misinterpreted as timeouts, please log in as root
    and temporarily modify /proc/sys/kernel/core_pattern, like so:

    echo core >/proc/sys/kernel/core_pattern
```

After following these instructions, we could successfully run the 8 parallel instances of AFLNet. Each instance has a separate status screen which can be used to follow the fuzzing-progress in real time, as shown in Figure 9.

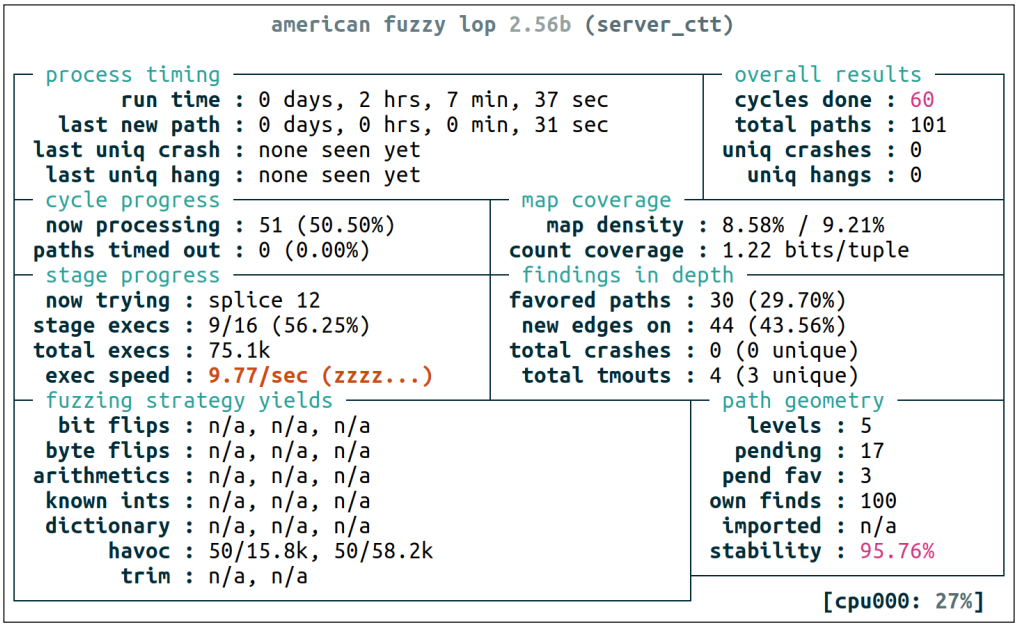

Figure 9: AFL(Net) status screen

Furthermore, the `afl-whatsup` tool can be used to view the combined progress of all fuzzers:

```
$ afl-whatsup out/temp_sync/
status check tool for afl-fuzz by <lcamtuf@google.com>

Individual fuzzers
==================

>>> fuzzer0 (0 days, 0 hrs) <<<

  cycle 1, lifetime speed 9 execs/sec, path 0/26 (0%)
  pending 2/26, coverage 9.05%, no crashes yet

...

Summary stats
=============

       Fuzzers alive : 8
      Total run time : 0 days, 1 hours
         Total execs : 0 million
    Cumulative speed : 72 execs/sec
       Pending paths : 76 faves, 572 total
  Pending per fuzzer : 9 faves, 71 total (on average)
       Crashes found : 0 locally unique
```

Since AFLNet does not have a fixed number of test cases, it runs indefinitely unless stopped. We fuzzed open62541 for 48 hours. The legacy OPC UA ANSI-C Stack and FreeOpcUa were fuzzed for 24 hours.

5.3 Fuzzing results

		Crashes		**Hangs**			
Implementation	**Lang.**	Total	Reproducible	Total	Reproducible	**Exec/s**	**Runtime**
legacy OPC UA ANSI-C Stack	C	8	8	0	0	$\pm 20 \cdot 8$	48 hours
open62541	C	68	0	14	0	$\pm 8 \cdot 8$	24 hours
FreeOpcUa	C++	43	43	9	0	$\pm 8 \cdot 8$	24 hours

Figure 10: Results of grey-box fuzzing using AFLNet

As can be seen in the table in Figure 10, we found several crashes and hangs in various OPC UA implementations using AFLNet. Some of these crashes are reproducible. In order to test the reproducibility of a crash, we use `aflnet-replay`. This tool takes a crash from the `replayable-crashes` output by AFLNet and sends it to an already running instance of the SUT. In section Section 5.4, we explain how we analyzed the crashes found by AFLNet. In sections Sections 5.4.1 and 5.4.2, we analyze the crashes that were found in FreeOpcUa and the legacy OPC UA ANSI-C Stack.

5.4 Analyzing crashes found by AFLNet

After finding a crash, AFLNet stores the data that caused the SUT to crash in the `replayable-crashes` folder. We analyze the found crashes using the following steps:

1. Verify if the crash is reproducible using `aflnet-replay`.
2. Minimize the crash data.
3. Debug the crash using GDB.
4. Using the output from GDB, analyze the source code to find the root cause of the crash.

To use `aflnet-replay`, we have to start the SUT beforehand. We run the SUT with a memory limit of 50 MB, which is the same memory limit as AFLNet uses by default, by running the SUT as follows[23]:

```
$ (ulimit -Sv $[50 << 10]; <SUT binary>)
```

In a different terminal, we run `aflnet-replay` as follows:

```
$ aflnet-replay <file> OPCUA 4840
```

Finally, we check if the SUT is still running and if it has output any logs. In case the SUT crashed, we want to continue the analysis. We want to minimize the crash case and remove as much unnecessary data from it as possible to make it easier to diagnose. Unfortunately, AFLNet does not have a tool to do this automatically, unlike AFL's `afl-cmin`, so we have to minimize crashes manually. We can minimize the crashes manually using `dd`:

```
$ dd if=crash of=crash_trimmed bs=1 count=<count>
```

Our goal is to find the smallest value for `count` that still crashes the SUT. We manually modify the parameter and re-test after each change using `aflnet-replay` until we find the minimized the crash data maximally such that it still crashes the SUT.

Once we have minimized the crash data, we run the SUT in the debugger GDB and replay the crash again. GDB is able to give us more information about why the SUT crashed. Most notably, GDB is able to print a backtrace of the crash by using the `bt` command. A backtrace contains the chain of function calls that led to the crash, including parameter values. We use the backtrace, combined with the source code of the SUT, to find the root cause of the crash.

5.4.1 Analyzing the FreeOpcUa crashes

AFLNet found a total of 48 reproducible crashes, as can be seen in Figure 10. These 48 crashes result in two types of behavior.

- The first type of crash only crashes FreeOpcUa when a memory limit of 50MB is set. Otherwise, it causes FreeOpcUa to get stuck in an infinite loop. This loop causes the CPU usage of the FreeOpcUa example server to top out. However, new connections are still accepted until a connection limit is hit. We decided not to further analyze this crash during this thesis, since it only crashes the FreeOpcUa server under unrealistic conditions.

[23] See '7) Interpreting output' in the documentation of AFL: `https://lcamtuf.coredump.cx/afl/README.txt`

- The second type of crash results in the FreeOpcUa example server binary terminating due to an uncaught error, stating that the buffer size given to `InputFromBuffer` was invalid. A screenshot of this behavior can be seen in Figure 11. Since this crashes the full server and thus prevents future connections, we chose to analyze this crash further.

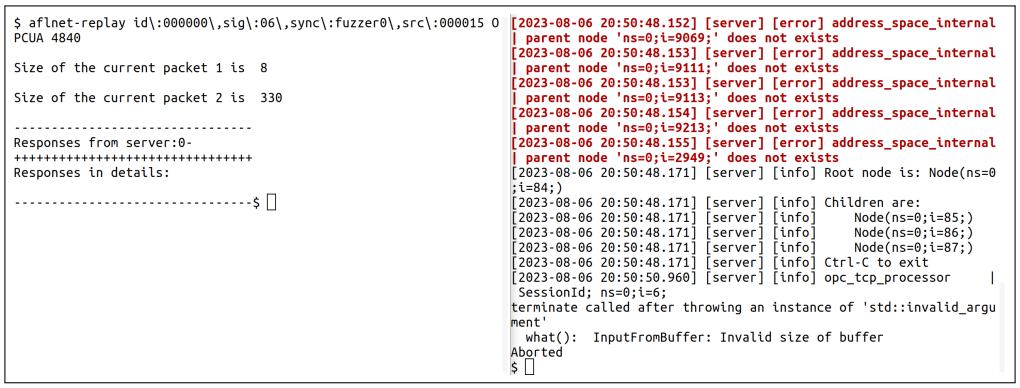

Figure 11: Replaying a crash on a FreeOpcUa server
On the left side, we use `aflnet-replay` to replay the crash. On the right side, the FreeOpcUa example server was running.

The first thing we did was minimize the crash data. Originally, the crash data that AFLNet output was 1214 bytes long. By manually minimizing it as described in Section 5.4, we reduced the size of the crash data to 9 bytes. The following data crashes the FreeOpcUa example server:

```
0800 0000 4845 4c46 08                    ....HELF.
```

1. The first 4 bytes, `0800 0000`, is the length of the message as used by `aflnet-replay`. This is not relevant to the crash.

2. The next 3 bytes, `4845 4c` describe the message type, `HEL` or Hello. The structure of this message type has been given in Appendix A.2.2

3. The next byte `46` describes the chunking type `F`. For the Hello message, this value is reserved and should be hard-coded to `F`.

4. Finally, `08` describes the length of the current OPC UA message in little-endian. Since `00`-bytes are assumed after the crash file ends, this length-value is interpreted by FreeOpcUa as `08 00 00 00`, or 8 bytes.

Now, let's use GDB to analyze the crash. We use `$ gdb ./example_server` to open FreeOpcUa's example server in GDB and then run it using `(gdb) run`. We then use `aflnet-replay` to replay the minimized crash file and crash the binary. We can now print a backtrace of line of code that resulted in the crash by using `(gdb) bt`. The most important parts of this backtrace printed by GDB can be seen below:

```
#0  __GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:50
...
#6  0x00007ffff6e114e2 in OpcUa::InputFromBuffer::InputFromBuffer (this=<
    optimized out>,
    buf=0x7ffff0001f00 "HELF\b", bufSize=0)
    at /home/ubuntu/Documents/Implementations/freeopcua/src/protocol/
        input_from_buffer.cpp:35
#7  0x00007ffff741c182 in (anonymous namespace)::OpcTcpConnection::
    ProcessMessage (
    this=0x7ffff0000e20, type=OpcUa::Binary::MT_HELLO, error=...,
        bytesTransferred=0)
    at /home/ubuntu/Documents/Implementations/freeopcua/src/server/
        opc_tcp_async.cpp:238
#8  0x00007ffff74197d4 in (anonymous namespace)::OpcTcpConnection::
    ProcessHeader(boost::system::error_code const&, unsigned long)::$_1::
    operator()(boost::system::error_code const&, unsigned long) const
    (this=0x7ffff66c1ba0, error=..., bytesTransferred=140737327972363)
at /home/ubuntu/Documents/Implementations/freeopcua/src/server/opc_tcp_async.
    cpp:220
...
#20 0x00007ffff733efd6 in boost::asio::detail::scheduler_operation::complete (
    this=0x7ffff0005210,
    owner=0x490380, ec=..., bytes_transferred=0)
...
#30 0x00007ffff6afbdf4 in ?? () from /lib/x86_64-linux-gnu/libstdc++.so.6
#31 0x00007ffff7f98609 in start_thread (arg=<optimized out>) at pthread_create.
    c:477
#32 0x00007ffff67e8133 in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S
    :95
```

We can see that `InputFromBuffer` is called with the parameters buffer `"HELF\b"` and buffer size 0, and that an error is thrown afterwards. Looking at `input_from_buffer.cpp` from FreeOpcUa's source code reveals that the error is thrown due to the buffer size being 0:

```
InputFromBuffer::InputFromBuffer(const char * buf, std::size_t bufSize)
: Buffer(buf)
, Size(bufSize)
{
  if (!Buffer) { throw std::invalid_argument("InputFromBuffer: Input buffer
      cannot be empty."); }

  if (!bufSize) { throw std::invalid_argument("InputFromBuffer: Invalid size of
      buffer"); }
}
```

By further analyzing the source files referred to in the backtrace, we eventually find that the buffer size of 0 originates from the fact that FreeOpcUa reads the OPC UA header and contents separately. First, FreeOpcUa reads the header from the network socket in the `OpcTcpConnection::ReadNextData` function. It does this by reading the first 8 bytes of the socket, which corresponds with the header size of OPC UA. It passes the read data on to the `ProcessHeader` function, which, after processing the header, tries to read the message after the header. It determines the number of bytes it needs to read by subtracting the total size of the message, determined from the header, by the hardcoded header size.

The problem is that, according to the header in the crash data, the message is 8 bytes long. Since an OPC UA header is also 8 bytes long, this means that FreeOpcUa will attempt to read 0 bytes from the message. When it tries to create a buffer of 0 bytes, the error will thus be thrown and result in a crash.

This crash should be fixable by catching the error higher-up in the code. Preferred behavior should be that FreeOpcUa sends an error message to the client and gracefully closes the connection, while still allowing for future connections. Unfortunately, we were not able to report the issue, as the maintainers of FreeOpcUa do not have any contact details for this. In May 2022, another security researcher opened a GitHub issue asking

to whom he could responsibly report a security issue[24]. As of August 2023, this issue has no replies.

5.4.2 Analyzing the legacy OPC UA ANSI-C crash

AFLNet found a single type of crash for the legacy OPC UA ANSI-C implementation. This crash results in a segmentation fault, as can be seen in Figure 12.

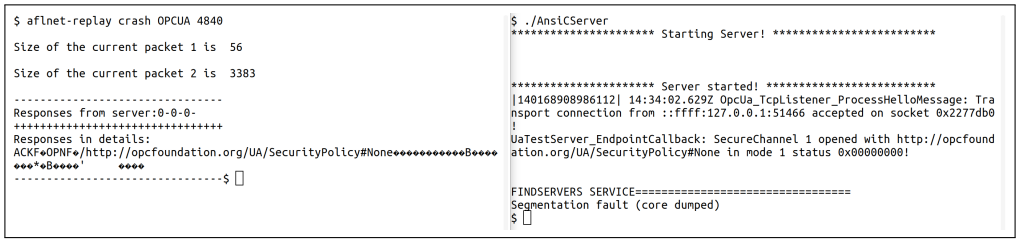

Figure 12: Replaying a crash on a legacy OPC UA ANSI-C Stack server
On the left side, we use `aflnet-replay` to replay the crash. On the right side, the legacy OPC UA ANSI-C Stack example server was running.

After manually minimizing this crash using the method described in Section 5.4, we found the following backtrace after crashing the server while it was running in GDB. The important parts of the backtrace are shown below:

```
#0  __strncmp_avx2 ()
#1  0x0000000000559aaa in OpcUa_P_String_strncmp (string1=0x0,
    string2=0x56c269 "Nano_Server", uiLength=<optimized out>)
    at /home/ubuntu/Documents/Implementations/UA-AnsiC-Legacy-1.04.342/UA-AnsiC
        -Legacy-1.04.342/Stack/platforms/linux/opcua_p_string.c:87
#2  0x000000000042a310 in my_FindServers (a_hEndpoint=<optimized out>,
    a_hContext=<optimized out>, a_pRequestHeader=0x7ffff0021950,
    a_pEndpointUrl=<optimized out>, a_nNoOfLocaleIds=<optimized out>,
    a_pLocaleIds=<optimized out>, a_nNoOfServerUris=1,
    a_pServerUris=0x7ffff0003490, a_pResponseHeader=0x7ffff00114e0,
    a_pNoOfServers=0x7ffff0011580, a_pServers=0x7ffff0011588)
    at /home/ubuntu/Documents/Implementations/UA-AnsiC-Legacy-1.04.342/UA-AnsiC
        -Legacy-1.04.342/AnsiCSample/ansicservermain.c:498
#3  0x000000000043f25c in OpcUa_Server_BeginFindServers
#4  0x000000000043db9e in OpcUa_Endpoint_BeginProcessRequest
#5  OpcUa_Endpoint_OnNotify
#6  0x0000000000456a16 in OpcUa_SecureListener_ProcessSessionCallRequest
#7  0x000000000045762b in OpcUa_SecureListener_ProcessRequest
#8  0x0000000000450998 in OpcUa_SecureListener_OnNotify
#9  0x000000000053e836 in OpcUa_TcpListener_ProcessRequest
#10 OpcUa_TcpListener_ReadEventHandler
#11 0x000000000053dc7f in OpcUa_TcpListener_EventCallback
#12 0x00000000005536ed in OpcUa_Socket_HandleEvent
#13 0x00000000005551e9 in OpcUa_P_Socket_HandleFdSet
#14 0x0000000000554857 in OpcUa_P_SocketManager_ServeLoopInternal
#15 0x0000000000551173 in OpcUa_P_SocketManager_ServerLoopThread
#16 0x000000000055a006 in pthread_start
#17 0x00007ffff7f98609 in start_thread
#18 0x00007ffff7b54133 in clone ()
```

At #1 in the backtrace, we find that `OpcUa_P_String_strncmp` is called with the first string being `null` and the second string being `"Nano_Server"`. This function calls the C function `strncmp`.

[24] `https://github.com/FreeOpcUa/freeopcua/issues/391`

```
/*============================================================================
* Compare two OpcUa_Strings Case Sensitive
*===========================================================================*/
OpcUa_Int32 OPCUA_DLLCALL OpcUa_P_String_strncmp(OpcUa_StringA string1,
    OpcUa_StringA string2, OpcUa_UInt32 uiLength)
{
    return (OpcUa_Int32)strncmp(string1, string2, uiLength);
}
```

Using `strncmp` with a `null` string on either side is considered undefined behavior, and causes a segmentation fault in this case. #2 in the backtrace contains the function that requested the `strncmp`, `my_FindServers`. This is a function defined in the example server of the legacy OPC UA ANSI-C Stack. Within the following code, the string compare is called:

```
    for(i=0;i<a_nNoOfServerUris;i++)
    {
        if(((OpcUa_Port_CallTable*)UaTestServer_g_PlatformLayerHandle)->StrnCmp
            (OpcUa_String_GetRawString(a_pServerUris+i),"Nano_Server",((
            OpcUa_Port_CallTable*)UaTestServer_g_PlatformLayerHandle)->StrLen("
            Nano_Server"))==0)
            break;
        *a_pNoOfServers=0;
        *a_pServers=OpcUa_Null;
        OpcUa_GotoError
    }
```

This code is run when a FindServers request is received by the server. It loops over the server URIs inside the request and checks if the server name matches the one of the example server, namely `"Nano_Server"`.

By analyzing the crash data using Wireshark, we can more easily see what the fields are within this data. As can be seen in Figure 13, the `EndpointUrl` value is indeed `null`.

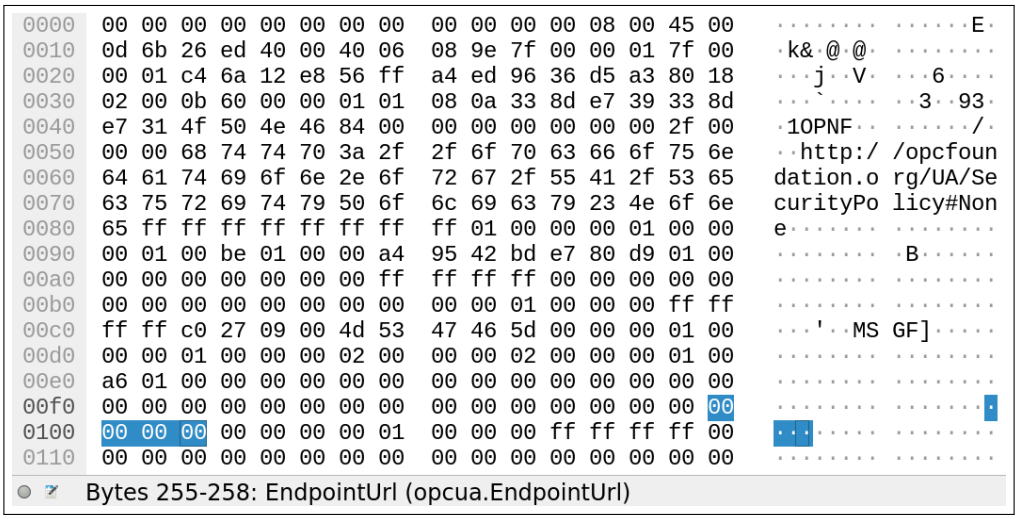

Figure 13: Analyzing OPC UA packets using Wireshark

Since this bug originates from the example server provided with the legacy OPC UA ANSI-C Stack and not the main code of the stack itself, and since this stack is no longer directly supported nor available, we have chosen not to report this issue. While parts of the legacy OPC UA ANSI-C Stack are used within the actively maintained OPC Foundation LDS (Local Discovery Server) Stack[25], the example server is not used there.

[25] `https://github.com/OPCFoundation/UA-LDS`

5.5 Conclusions

While performing this experiment, we expected that we would find crashes more effectively than the Boofuzz experiment performed in Chapter 4. AFLNet was able to find multiple reproducible crashes in both the legacy OPC UA ANSI-C Stack and FreeOpcUa, while Boofuzz was able to find none. Part of this could be attributed to the fact that AFLNet explores the state space of OPC UA more extensively, as can be seen by comparing the state coverage of Boofuzz described in Section 4.2 and the state coverage of AFLNet described in Section 5.2.3. Furthermore, AFLNet is 'smarter' in exploring the state space of the SUTs, as it uses instrumentation to determine and prioritize the next messages it is going to send, while Boofuzz generates the same messages for each run. Thus, we could conclude that, when comparing AFLNet to Boofuzz, AFLNet is indeed able to find crashes more effectively than the Boofuzz setup used in Chapter 4.

Setup. As described in Section 5.1, AFLNet does not have built in support for the OPC UA protocol. Therefore, we had to implement this ourselves by modifying the source code of AFLNet. The documentation of AFLNet, alongside the countless examples of other implemented protocols, made this simple to do.

AFLNet requires seed inputs based on actual communication. We can generate these seed inputs by capturing network communication between a client and server and extracting the individual traces from this communication. This requires significantly less effort than writing both the message and state grammar of the SUT from the ground up.

The tooling provided by AFLNet is useful and well documented, as described in Section 5.2.4. The documentation of both AFL and AFLNet are clear, providing examples and advice for running the fuzzer and resolving problems. The main user interface, as seen in Figure 9 provides a clear overview of the fuzzing process. `afl-whatsup` gives useful statistics for a currently running `aflnet-replay` allows for quickly replaying and verifying found crashes.

AFL has a tool named `afl-cmin` which minimizes found crashes in order to aid debugging. Due to the fact that AFL inputs crashes to the SUT as a file, and AFLNet does so via the network, `afl-cmin` is not compatible with the SUTs we are fuzzing, and would thus require a separate tool in order to do this. Unfortunately, AFLNet does not have an automated way of minimizing crashes.

Reproducibility. The AFLNet setup we made for this thesis directly ran in a virtual machine without using containers. The setup-script for AFLNet provided in Appendix E and the setup-scripts for the OPC UA implementations we tested with AFLNet in Appendix D should allow for the experiments to be accurately reproducible on other machines in the future. However, they have only been tested on Ubuntu 20.04, and do not work on Ubuntu 22.10 or higher without further modifications. Containerizing the AFLNet setup should resolve the dependency on a specific operating system version.

Extensibility. Unlike the Boofuzz experiment performed in Chapter 4, where we can build and install the SUTs with few modifications, AFLNet requires us to build the SUTs with instrumentation enabled, such that AFLNet can retrieve direct feedback from the SUT as explained in Section 2.2.4.

For the three implementations we tested in this chapter, this proved to be simple, as this could be done by changing the compiler to `afl-clang-fast(++)`. However, it is an additional step that needs to be performed. Furthermore, the instrumentation that has to be added to the SUT is only compatible with C and C++ programs. This limits the number of OPC UA implementations we can test. Performing grey-box fuzzing with

different tooling on other implementations has been left as future work and is further discussed in Section 6.5.

Performance. The performance of AFLNet on the tested OPC UA implementations resulted in roughly 8 executions per second per core. This is a large improvement over the 1 execution per second average that was achieved during the Boofuzz experiment as shown in Section 4.6.

The capability of AFL and AFLNet to fuzz in parallel increased this performance even more, by roughly the number of cores the system has, thus achieving results of up to 64 executions per second. Parallel fuzzing requires that the SUT each AFLNet instance fuzzes uses a different port. As described in Section 5.2, we achieved this by modifying the source code of the SUT to either add a parameter to configure the server port, or to make different binaries with different hard-coded ports.

While the latter method is simpler, it does not allow for scalability should a larger system be used in the future. As shown in Section 5.2.4, parallel fuzzing requires us to run a separate command for each instance of AFLNet we want to run. This once again gives issues with scalability. While fuzzing like this is feasible for the 8-core system we used, automating this would result in improved scalability, allowing us to re-run these experiments more easily on systems with higher core-counts.

Unlike Boofuzz, AFLNet does not have a fixed number of testcases. Instead, AFLNet tries to find and explore any path it finds given it has unlimited time, prioritizing more interesting paths first. This means that AFLNet does not stop by itself. We limited the time for open62541 to 48 hours and the time for the legacy OPC UA ANSI-C Stack and FreeOpcUa to 24 hours.

6 Future Work

In this chapter, we discuss ideas for future work on fuzzing OPC UA implementations.

6.1 Message timeouts

As discussed in the results of the Boofuzz experiment in Section 4.5, Boofuzz used 5 second timeouts for both sending and receiving messages from the SUT. These timeouts are used to ensure that these messages properly arrive at their destination. However, it is likely that these delays can be decreased significantly without resulting in any degraded results. AFLNet, for example, fuzzed open62541 using a maximum receiving timeout of about 0.2 seconds.

6.2 Fuzzing more OPC UA message types

In section Section 2.3.3, we discussed the nine message types that the Boofuzz setup by BSI fuzzes. Furthermore, in Section 4.2 for black-box fuzzing and Section 5.2.3 for grey-box fuzzing, we discuss the state coverage of the inputs of both methods. By comparing these with the list of session sets the OPC UA specification contains, we find that there are multiple session sets and corresponding message types we have not fuzzed completely. Service sets we did not completely fuzz include the NodeManagement, Attribute and Subscription service sets.

6.3 Fuzzing more OPC UA implementations

In this thesis, we only fuzzed the OPC UA implementations listed in Section 3.4. We limited ourselves to only fuzz open source OPC UA implementations. However, as has been documented by Erba et al. [8], many more open source OPC UA implementations exist.

Most OPC UA implementations we tested were easy to setup and configure for fuzzing. For these OPC UA implementations, it was simple to modify their configurations to disable encryption and build them with either sanitizers or instrumentation when needed. We discovered that some implementations, such as S2OPC [27], do not have a simple toggle or setting to disable this and would require more work in order to make them suitable for fuzzing.

It is very likely that more OPC UA implementations will be developed in the future. Furthermore, if we would also consider closed source and commercial implementations, the number of potential OPC UA implementations would increase drastically.

6.4 Improving performance of network-based fuzzing

We only used network based fuzzers in both experiments we performed. However, the network communication that needs to happen between the fuzzer and the SUT introduces a high amount of latency. In order to improve fuzzing performance, the SUT could be rewritten to read from local files instead of network sockets.

6.5 Using different fuzzers

As described in Section 2.2, various types of fuzzing exist. Furthermore, there are many different tools that use different methods to perform fuzzing. As such, these tools might achieve different levels of coverage, which could allow us to find more bugs. Other tools might also result in performance improvements compared to Boofuzz and AFLNet.

An interesting fuzzer to consider is SGFuzz[1]. The authors of SGFuzz claim that it covers code of the SUT X times faster than AFLNet. However, SGFuzz is newer and less widely used than AFLNet. Furthermore, the documentation of SGFuzz appears to be less clear than that of AFLNet. It is likely that it is more difficult to set-up SGFuzz, but the potential performance increase makes it interesting to consider.

We have also seen that some fuzzers have limitations on which programs they can fuzz. For example, AFL and AFLNet are only able to instrument C and C++ binaries, which limits the number of potential SUTs we can fuzz. For AFL, various forks exist that allow it to function with programs written in other languages, such as sharpfuzz[26] for C# and Kelenci[27] for Java.

6.6 Improving code coverage

During our research, we mainly compared Boofuzz and AFLNet based on performance metrics and the number of reproducible crashes they found. A different method to compare the efficiency of different fuzzers is to measure the code coverage provided by the fuzzers on each OPC UA implementation.

Furthermore, it is very likely that the example server applications provided with the OPC UA implementations we fuzzed do not give access to the full feature set of the OPC UA implementation. By measuring code coverage on the OPC UA implementation itself, we are able to determine how much of the implementation's code can be covered by the fuzzer. Using the found code coverage, modifications or extensions can be made to the fuzzed example server to give the fuzzer access to more code of the SUT, which should in turn improve the efficiency of the fuzzer.

[26] `https://github.com/Metalnem/sharpfuzz`
[27] `https://github.com/isstac/kelinci`

7 Conclusions

In this thesis, we aimed to answer the research question *How do we effectively fuzz open source implementations of the OPC UA protocol?*. We performed fuzzing on 7 OPC UA implementations using the black-box fuzzer Boofuzz using an setup based on one by BSI and fuzzed 3 C/C++ OPC UA implementations using the grey-box fuzzer AFLNet. We found that in this thesis, grey-box fuzzing OPC UA implementations using AFLNet was more effective than black-box fuzzing using Boofuzz. We consider some differences between Boofuzz and AFLNet that led us to this conclusion:

- **Setup.** In Section 4.2, we found that the setup made by BSI was simple to setup and reset due to the usage of Python scripts which set up Docker containers for fuzzing. However, this setup also made it more difficult to debug any errors that occurred. The AFLNet setup we made does not use any Docker containers, which could potentially make it harder to setup on different systems. We found that out of the two implementations, the documentation of AFLNet was slightly clearer, with AFLNet itself often presenting steps that could be taken to solve a problem as described in Section 5.2.4. Boofuzz, on the other hand, would require us to search through their official documentation to do this. Finally, we found that the tooling of AFLNet was much better, as it has a clear command line interface, has support for parallel fuzzing and can easily replay crashes trough `aflnet-replay`.

- **Reproducibility.** The Boofuzz setup by BSI had several problems with reproducibility, as can be seen in Section 4.5. For node-opcua, we had to fix mismatched version numbers in the installation script to ensure the example server could run in the fuzzing container. We could not get the .NET Standard Stack to run at all due to dependency errors. With the setup of the AFLNet experiments described in Section 5.2.2, we ensured to hard-code all versions and aimed to make the type of system we can install it on as clear as possible. A future step could be setting up Docker containers to perform AFLNet fuzzing in.

- **Extensibility.** The Boofuzz setup is simple to extend. Adding support for a new OPC UA implementation can be done by adding an installation script for this OPC UA implementatino to the setup, as described in section Section 4.3. This script downloads, compiles and copies the server binary to the correct path so Boofuzz can use it. AFLNet, on the other hand, requires us to build the OPC UA implementations with instrumentation, which was described in Section 5.2.2. Furthermore, AFLNet is limited to C/C++ binaries, limiting the amount of testable OPC UA implementations. The input message- and state-grammar of Boofuzz, on the other hand, is more difficult to extend compared to AFLNet, where we can add new input seeds by capturing different kinds of communication between a client and a server when we want to test different kinds of messages. For both Boofuzz and AFLNet, it is important that the fuzzed OPC UA implementations have encryption and other configuration settings such as session limits disabled.

- **Performance.** In general, fuzzing a binary over network sockets introduces a large performance overhead. The performance from the Boofuzz setup was very bad, varying between 2 and 0.5 executions per second depending on the tested OPC UA implementation, as shown in Section 4.5. A lot of test-cases from Boofuzz hit Boofuzzes timeout of exactly 10 seconds, which caused this low fuzzing performance. In comparison, AFLNet was much faster, as can be seen in Section 5.3. In the worst case, each core fuzzed the SUTs with roughly 8 executions per second. Because we ran AFLNet in parallel on eight cores, this resulted in 64 executions per second in total.

No reproducible crashes were found using Boofuzz, while AFLNet found three types of reproducible crashes, as described in Section 5.4. Two crashes were found in FreeOpcUa, and one crash was found in the legacy ANSI-C OPC UA implementation:

- The crash we found in the legacy ANSI-C OPC UA implementation was caused by incorrectly checked parameters to the string comparison function `strncmp`, causing a crash when one of the strings was set to `null` via user input.
- One crash found in FreeOpcUa only occurred in low memory situations. In high memory situations, the same input caused the server to run into an infinite loop, causing high CPU usage, but did not prevent new connections until a connection limit was hit.
- The second crash found in FreeOpcUa occurred because of an uncaught exception that occurred when a buffer of size 0 was requested, which was possible by setting the size variable within an OPC UA message header to 8. Since FreeOpcUa subtracts this number from the header size which is also 8, this resulted in FreeOpcUa trying create a buffer of size 0, throwing an error and crashing the OPC UA implementation.

References

[1] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. "Stateful Greybox Fuzzing". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3255–3272. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/ba.

[2] Pavel Cheremushkin. *Practical example of fuzzzing OPC UA applications*. Kaspersky Lab ICS CERT, 2020. URL: https://ics-cert.kaspersky.com/publications/reports/2020/10/19/practical-example-of-fuzzing-opc-ua-applications/.

[3] Pavel Cheremushkin and Sergey Temnikov. *OPC UA security analysis*. Kaspersky Lab ICS CERT, 2018. URL: https://ics-cert.kaspersky.com/publications/reports/2018/05/10/opc-ua-security-analysis/.

[4] Markus Dahlmanns, Johannes Lohmöller, Ina Berenice Fink, Jan Pennekamp, Klaus Wehrle, and Martin Henze. "Easing the Conscience with OPC UA: An Internet-Wide Study on Insecure Deployments". In: *Proceedings of the ACM Internet Measurement Conference*. IMC '20. Virtual Event, USA: ACM, 2020, pp. 101–110. DOI: 10.1145/3419394.3423666.

[5] Damm, Gappmeier, Zugfil, Plöb, Fiat, and Störtkuhl. *OPC UA Security Analysis*[28]. Bundesamt für Sicherheit in der Informationstechnik, 2017. URL: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/OPCUA/OPCUA.html.

[6] Cristian Daniele, Seyed Behnam Andarzian, and Erik Poll. *Fuzzers for stateful systems: Survey and Research Directions*. 2023. DOI: 10.48550/ARXIV.2301.02490.

[7] Johannes vom Dorp, Sven Merschjohann, David Meier, Florian Patzer, Markus Karch, and Christian Haas. *OPC UA Security Analysis*[29]. Bundesamt für Sicherheit in der Informationstechnik, 2022. URL: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/OPCUA/OPCUA_2022_EN.pdf.

[8] Alessandro Erba, Anne Müller, and Nils Ole Tippenhauer. "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*. CPSIoTSec '22. Los Angeles, CA, USA: ACM, 2022, pp. 1–13. DOI: 10.1145/3560826.3563380.

[9] Eclipse Foundation. *Eclipse Milo*. OPC UA Implementation. URL: https://www.eclipse.org/milo (visited on May 21, 2023).

[10] OPC Foundation. *OPC UA ANSIC! The official UA ANSI-C Stack and Sample Applications from the OPC Foundation*. OPC UA Implementation. URL: http://opcfoundation.github.io/UA-AnsiC-Legacy (visited on May 22, 2023).

[11] OPC Foundation. *OPC Unified Architecture .NET Standard*. OPC UA Implementation. URL: https://github.com/OPCFoundation/UA-.NETStandard (visited on May 21, 2023).

[12] FreeOpcUa. *python-opcua*. OPC UA Implementation. URL: https://github.com/FreeOpcUa/python-opcua (visited on May 21, 2023).

[28] German version of report from 2017 retrieved from https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/OPCUA/OPCUA.pdf

[29] German version of report from 2022 retrieved from https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/OPCUA/OPCUA_2022.html

[13] *FreeOpcUA: Open Source C++ OPC-UA Server and Client Library*. OPC UA Implementation. URL: https://github.com/FreeOpcUa/freeopcua (visited on Aug. 6, 2023).

[14] Patrice Godefroid. "Fuzzing: Hack, Art, and Science". In: *Commun. ACM* 63.2 (Jan. 2020), pp. 70–76. DOI: 10.1145/3363824.

[15] Aki Helin. *Radamsa*. URL: https://gitlab.com/akihe/radamsa (visited on Apr. 13, 2023).

[16] Sam Hocevar. *zzuf*. 2007. URL: http://caca.zoy.org/wiki/zzuf (visited on Mar. 7, 2023).

[17] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. "The Art, Science, and Engineering of Fuzzing: A Survey". In: *IEEE Transactions on Software Engineering* 47.11 (2021), pp. 2312–2331. DOI: 10.1109/TSE.2019.2946563.

[18] Nikolas Mühlbauer, Erkin Kirdan, Marc-Oliver Pahl, and Georg Carle. "Open-Source OPC UA Security and Scalability". In: *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. 2020, pp. 262–269. DOI: 10.1109/ETFA46521.2020.9212091.

[19] *node-opcua*. OPC UA Implementation. URL: https://node-opcua.github.io (visited on May 21, 2023).

[20] *open62541*. OPC UA Implementation. URL: https://www.open62541.org (visited on May 21, 2023).

[21] *Peach Fuzzer*. 2004. URL: https://gitlab.com/peachtech/peach-fuzzer-community (visited on May 10, 2023).

[22] Joshua Pereyda. *boofuzz*. URL: https://github.com/jtpereyda/boofuzz (visited on Apr. 1, 2023).

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

[24] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. "AddressSanitizer: A Fast Address Sanity Checker". In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, June 2012, pp. 309–318. ISBN: 978931971935.

[25] Konstantin Serebryany and Timur Iskhodzhanov. "ThreadSanitizer: Data Race Detection in Practice". In: *Proceedings of the Workshop on Binary Instrumentation and Applications*. WBIA '09. New York, New York, USA: ACM, 2009, pp. 62–71. DOI: 10.1145/1791194.1791203.

[26] Evgeniy Stepanov and Konstantin Serebryany. "MemorySanitizer: Fast detector of uninitialized memory use in C++". In: *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2015, pp. 46–55. DOI: 10.1109/CGO.2015.7054186.

[27] Systerel. *S2OPC*. OPC UA Implementation. URL: https://www.s2opc.com (visited on May 21, 2023).

[28] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. "Scheduling Black-Box Mutational Fuzzing". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS '13. Berlin, Germany: ACM, 2013, pp. 511–522. DOI: 10.1145/2508859.2516736.

[29] Yingchao Yu, Zuoning Chen, Shuitao Gan, and Xiaofeng Wang. “SGPFuzzer: A State-Driven Smart Graybox Protocol Fuzzer for Network Protocol Implementations”. In: *IEEE Access* 8 (2020), pp. 198668–198678. DOI: 10.1109/ACCESS.2020.3025037.

[30] Michał Zalewski. *Technical "whitepaper" for afl-fuzz*. Tech. rep. URL: https://lcamtuf.coredump.cx/afl/technical_details.txt.

Glossary

ASAN AddressSanitizer.

ICS Industrial Control Systems.

MSAN MemorySanitizer.

OPC UA Open Platform Communications Unified Architecture.

SUT System Under Test.

TSAN ThreadSanitizer.

UBSAN UndefinedBehaviorSanitizer.

Appendices

Appendix A OPC UA message structures

This appendix contains several examples with diagrams for how OPC UA messages are structured. Section Section A.1 contains information about the string type used by OPC UA and section Section A.2 contains examples of message types OPC UA uses to initiate connections.

A.1 Types

The OPC UA specification lists several built in types that can be used within messages[30]. The types that are relevant to the messages that we fuzzed are listed in this section.

A.1.1 String type

The table in Figure 14 contains the structure of the string type.

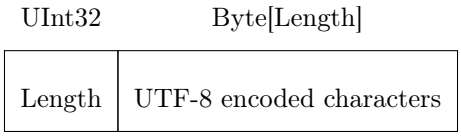

Figure 14: String type structure

A.2 Connection protocol

Connection protocol messages[31] are responsible for initiating a communication channel between an OPC UA client and server.

A.2.1 Message header

The OPC UA message header, as described in section 7.1.2.2 of the OPC UA specification[32], is prefixed to all messages sent by the OPC UA protocol. Note that the MessageSize field contains the message size including the 8 bytes of the header. The table in Figure 15 contains the structure for message headers.

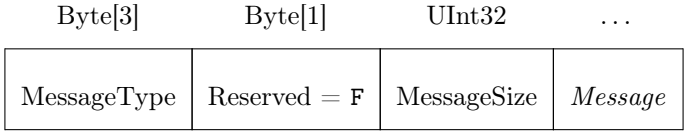

Figure 15: Message header structure

[30] `https://reference.opcfoundation.org/Core/Part6/v105/docs/5.2` (OPC UA Binary)
[31] `https://reference.opcfoundation.org/Core/Part6/v105/docs/7.1` (OPC UA Connection Protocol)
[32] `https://reference.opcfoundation.org/Core/Part6/v105/docs/7.1.2.2` (Message Header)

A.2.2 Hello message

The hello message is sent from the client to the server in order to initiate a connection. It contains several fields which state the capabilities of the client, such as the sizes of the send and receive buffers. The table in Figure 16 contains the packet structure of the Hello message (`HEL`).

...	UInt32	UInt32	UInt32	UInt32	UInt32	String
Header	ProtocolVersion	ReceiveBufferSize	SendBufferSize	MaxMessageSize	MaxChunkCount	EndpointUrl

Figure 16: Hello message packet structure

A.2.3 Error message

OPC UA error messages, as shown in section 7.1.2.5 of the OPC UA specification[33], contain a status code and an optional reason. A list of valid status codes can be found in section A.2 of the OPC UA specification[34]. The table in Figure 17 contains the structure of the error message (`ERR`).

...	UInt32	String
Header	Error	Reason

Figure 17: Error message packet structure

[33] `https://reference.opcfoundation.org/Core/Part6/v105/docs/7.1.2.5` (Error Message)
[34] `https://reference.opcfoundation.org/Core/Part6/v105/docs/A.2` (Status Codes)

Appendix B Building OPC UA implementations for Boofuzz

This appendix contains the `install.sh` scripts used to build several OPC UA implementations in the black-box fuzzing experiments performed in Chapter 4. Instructions on how to use these scripts to add support for new implementations to BSI's Boofuzz setup can be found in Section 4.3.

B.1 Legacy OPC UA ANSI-C Stack

This script assumes that the source code of the legacy OPC UA ANSI-C Stack is present in `targets/legacy-stack/UA-AnsiC-Legacy-1.04.342`.

```
#!/usr/bin/env bash

# Install dependencies
apt-get update && apt-get install libssl-dev cmake -y

cd targets/legacy-stack/UA-AnsiC-Legacy-1.04.342

# Build legacy OPC UA ANSI-C stack
mkdir build
cd build
cmake ..
cmake --build .

# Create symbolic link to target (example server)
cd ../../../../
ln -s targets/legacy-stack/UA-AnsiC-Legacy-1.04.342/build/bin/AnsiCServer /opt/app/target

exit 0
```

B.2 FreeOpcUa

```
#!/usr/bin/env bash

# Clone repository
git clone https://github.com/freeopcua/freeopcua.git
cd freeopcua
git checkout 6eac097

# Install cmake
apt-get update && apt-get install cmake -y

# Install dependencies using apt-get command listed in debian.soft
sed -i 's/libmbedtls-dev/libmbedtls-dev -y/g' debian.soft # Add -y to the end of apt-get command
./debian.soft

# Build freeopcua
mkdir build
cd build
cmake ..
make

# Copy example server
cp bin/example_server /opt/app/target

cd ../../

exit 0
```

Appendix C Building OPC UA implementations for AFLNet

This appendix contains the scripts we used to build several OPC UA implementations with instrumentation for usage with AFLNet in the grey-box fuzzing experiments performed in Chapter 5.

C.1 open62541

```
#!/bin/sh

# Install APT packages
apt-get install cmake -y

# Remove previous repository
rm -r open62541

# Clone repository and check out specific commit
git clone https://github.com/open62541/open62541.git
cd open62541
git checkout 46d0395
git submodule update --init --recursive

# Copy modified server_ctt.c that allows for anonymous authentication
cp ../server_ctt.c examples

# Build open62541 with examples
AFL_HARDEN=1 cmake -DCMAKE_C_COMPILER=afl-clang-fast -DCMAKE_CXX_COMPILER=afl-clang-fast++ -DBUILD_SHARED_LIBS=ON -DUA_BUILD_EXAMPLES=ON -DUA_ENABLE_ENCRYPTION=OFF ../open62541
make -j1
make install

# Copy server and build client
cd ..
cp open62541/bin/examples/server_ctt .
afl-clang-fast -std=c99 -DUA_ARCHITECTURE_POSIX client_anonymous.c -lopen62541 -o client_anonymous
```

C.2 Legacy OPC UA ANSI-C Stack

This script assumes that the source code of the legacy OPC UA ANSI-C Stack is present in `UA-AnsiC-Legacy-1.04.342`.

```
#!/usr/bin/env bash

# Install dependencies
apt-get update && apt-get install libssl-dev cmake -y

cd UA-AnsiC-Legacy-1.04.342

# Build legacy OPC UA ANSI-C stack
mkdir build
cd build
export AFL_HARDEN=1
cmake -DCMAKE_C_COMPILER=afl-clang-fast -DCMAKE_CXX_COMPILER=afl-clang-fast++ ..
cmake --build .
```

C.3 FreeOpcUa

```
1  #!/bin/sh
2
3  # Install cmake
4  apt-get update && apt-get install cmake -y
5
6  # Clone repository
7  git clone https://github.com/freeopcua/freeopcua.git
8  cd freeopcua
9  git checkout 6eac097
10
11 # Install dependencies using apt-get command listed in debian.soft
12 sed -i 's/libmbedtls-dev/libmbedtls-dev -y/g' debian.soft # Add -y to the end
       of apt-get command
13 ./debian.soft
14
15 # Build freeopcua
16 mkdir build
17 cd build
18 AFL_HARDEN=1 cmake -DCMAKE_C_COMPILER=afl-clang-fast -DCMAKE_CXX_COMPILER=afl-
       clang-fast++ ..
19 make
```

Appendix D Modifying AFLNet for fuzzing OPC UA

This appendix contains the source code of several modifications we made to AFLNet to make it support the OPC UA standard for the AFLNet experiment described in Chapter 5.

D.1 Extracting requests

The following function has been added to `aflnet.c`:

```
region_t* extract_requests_opcua(unsigned char* buf, unsigned int buf_size,
    unsigned int* region_count_ref)
{
  char *mem;
  unsigned int byte_count = 0;
  unsigned int mem_count = 0;
  unsigned int mem_size = 1024;
  unsigned int region_count = 0;
  region_t *regions = NULL;

  mem=(char *)ck_alloc(mem_size);

  unsigned int cur_start = 0;
  unsigned int cur_end = 0;
  while (byte_count < buf_size) {
    memcpy(&mem[mem_count], buf + byte_count++, 1);

    // Check if region is at least 8 bytes (OPC UA header size)
    if (mem_count >= 8) {
      // Bytes: 1-3: message type
      // Byte 4: chunk type
      // Bytes 5-8: message size

      // Extract message size from header
      u32 message_size = (u32)mem[4] << 0  |
                         (u32)mem[5] << 8  |
                         (u32)mem[6] << 16 |
                         (u32)mem[7] << 24;

      // Skip the payload, removing the OPC UA header (8) from the message size
      unsigned int bytes_to_skip = message_size - 8;

      unsigned int temp_count = 0;
      while ((byte_count < buf_size) && (temp_count < bytes_to_skip)) {
        byte_count++;
        cur_end++;
        temp_count++;
      }

      if (byte_count < buf_size) {
          byte_count--;
          cur_end--;
      }

      // Create one region
      region_count++;
      regions = (region_t *)ck_realloc(regions, region_count * sizeof(region_t)
          );
      regions[region_count - 1].start_byte = cur_start;
      regions[region_count - 1].end_byte = cur_end;
      regions[region_count - 1].state_sequence = NULL;
      regions[region_count - 1].state_count = 0;

      // Check if the last byte has been reached
      if (cur_end < buf_size - 1) {
        mem_count = 0;
        cur_start = cur_end + 1;
        cur_end = cur_start;
      }
    } else {
      mem_count++;
      cur_end++;

      // Check if the last byte has been reached
```

```
63          if (cur_end == buf_size - 1) {
64            region_count++;
65            regions = (region_t *)ck_realloc(regions, region_count * sizeof(
                  region_t));
66            regions[region_count - 1].start_byte = cur_start;
67            regions[region_count - 1].end_byte = cur_end;
68            regions[region_count - 1].state_sequence = NULL;
69            regions[region_count - 1].state_count = 0;
70            break;
71          }
72
73          if (mem_count == mem_size) {
74            mem_size = mem_size * 2;
75            mem=(char *)ck_realloc(mem, mem_size);
76          }
77        }
78      }
79      if (mem) ck_free(mem);
80
81      if ((region_count == 0) && (buf_size > 0)) {
82        regions = (region_t *)ck_realloc(regions, sizeof(region_t));
83        regions[0].start_byte = 0;
84        regions[0].end_byte = buf_size - 1;
85        regions[0].state_sequence = NULL;
86        regions[0].state_count = 0;
87
88        region_count = 1;
89      }
90
91      *region_count_ref = region_count;
92      return regions;
93  }
```

D.2 Extracting response codes

The following function has been added to `aflnet.c`:

```
1   unsigned int* extract_response_codes_opcua(unsigned char* buf, unsigned int
        buf_size, unsigned int* state_count_ref)
2   {
3     char *mem;
4     unsigned int byte_count = 0;
5     unsigned int mem_count = 0;
6     unsigned int mem_size = 1024;
7     unsigned char message_type;
8     unsigned int *state_sequence = NULL;
9     unsigned int state_count = 0;
10
11    mem=(char *)ck_alloc(mem_size);
12
13    //Add initial state
14    state_count++;
15    state_sequence = (unsigned int *)ck_realloc(state_sequence, state_count *
          sizeof(unsigned int));
16    state_sequence[state_count - 1] = 0;
17
18    while (byte_count < buf_size) {
19      memcpy(&mem[mem_count], buf + byte_count++, 1);
20
21      if (mem_count >= 8) {
22        // Bytes 0-2: message type
23        // Byte 3: chunk type
24        // Bytes 4-7: message size
25        // Bytes 8-11: error number
26
27        if ((mem[0] = 'E') && (mem[1] == 'R') && (mem[2] == 'R')) {
28          // If message type is ERR, we use the second byte from error code (
                endianness swapped), as this value has the most variance.
29          message_type = mem[10];
30        } else { // If message is not an ERR, interpret it as status code 0x00
31          message_type = 0x00;
32        }
33
34        // Extract message size from header
35        u32 message_size = (u32)mem[4] << 0  |
```

```
36                              (u32)mem[5] << 8  |
37                              (u32)mem[6] << 16 |
38                              (u32)mem[7] << 24;
39
40        // Skip the payload, removing the OPC UA header (8) from the message size
41        unsigned int bytes_to_skip = message_size - 8;
42        unsigned int temp_count = 0;
43        while ((byte_count < buf_size) && (temp_count < bytes_to_skip)) {
44          byte_count++;
45          temp_count++;
46        }
47
48        if (byte_count < buf_size) {
49            byte_count--;
50        }
51
52        // Add response code to array
53        unsigned int message_code = message_type;
54        state_count++;
55        state_sequence = (unsigned int *)ck_realloc(state_sequence, state_count *
              sizeof(unsigned int));
56        state_sequence[state_count - 1] = message_code;
57        mem_count = 0;
58      } else {
59        mem_count++;
60
61        if (mem_count == mem_size) {
62          mem_size = mem_size * 2;
63          mem=(char *)ck_realloc(mem, mem_size);
64        }
65      }
66    }
67    if (mem) ck_free(mem);
68
69    *state_count_ref = state_count;
70    return state_sequence;
71  }
```

D.3 Adding OPCUA option to the tooling

`afl-fuzz.c`

The following code adds `OPCUA` as an option to AFLNet's protocol parameter `-P`:

```
9073          } else if (!strcmp(optarg, "OPCUA")) {
9074            extract_requests = &extract_requests_opcua;
9075            extract_response_codes = &extract_response_codes_opcua;
9076          } else {
```

`aflnet-replay.c`

The following code adds `OPCUA` as an option in `aflnet-replay`:

```
50    else if (!strcmp(argv[2], "OPCUA")) extract_response_codes = &
          extract_response_codes_opcua;
```

Appendix E Installing AFLNet

This appendix contains the script used to install AFLNet on a virtual machine running Ubuntu 20.04. It is assumed that the modified source files `aflnet.h`, `aflnet.c`, `aflnet-replay.c` and `afl-fuzz.c` described in Appendix D are present in the same folder as the installation script in this appendix.

```
1  #!/bin/sh
2
3  # Install dependencies
4  sudo apt-get install git make llvm-dev clang graphviz-dev libgraphviz-dev
       libcap-dev -y
5
6  # Clone repository
7  git clone https://github.com/aflnet/aflnet.git aflnet
8  cd aflnet
9  git checkout 213c9cf
10
11 # Copy modified AFLNet source code
12 cp ../{aflnet.h,aflnet.c,aflnet-replay.c,afl-fuzz.c} .
13
14 # Build AFLNet
15 make clean all
16
17 export LLVM_CONFIG=llvm-config-10
18 cd llvm_mode
19 make
20
21 # Export environment variables
22 cd ../..
23 export AFLNET=$(pwd)/aflnet
24 export WORKDIR=$(pwd)
25 export PATH=$PATH:$AFLNET
26 export AFL_PATH=$AFLNET
```