DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2019

Exploring the Possibilities of Robustness Testing of CoAP Implementations Using Evolutionary Fuzzing

FREDRIK LILJEDAHL

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Exploring the Possibilities of Robustness Testing of CoAP Implementations Using Evolutionary Fuzzing

FREDRIK LILJEDAHL

Master in Computer Science
Date: September 27, 2019
Supervisor: Somayeh Aghanavesi
Examiner: Mads Dam
School of Electrical Engineering and Computer Science
Host company: Subset AB
Swedish title: Utforskande studie inom möjligheterna att testa robustheten av CoAP-implementationer med evolutionär fuzzing

Abstract

Internet of Things (IoT) is a widely used expression to denote the connection of physical objects in the internet. IoT devices are attractive targets for malicious actors as they are often deployed in large numbers with the same software. Such software is important to test in order to prevent malicious abuse. An effective robustness testing technique is called fuzzy testing (fuzzing) and involves automatically exposing software to a multitude of generated inputs to hopefully discover errors in the application before an attacker can exploit them. The Constrained Application Protocol (CoAP) is a relatively new application protocol designed for use in IoT devices whose implementations could contain vulnerabilities. Fuzzing of CoAP applications has been done with success in a previous study but there is room for improvements and further development of CoAP testing techniques.

In this report an exploratory study is performed on the possibility of using evolutionary algorithms to strengthen the effectiveness of regular fuzzing on CoAP server implementations. For this, an evolutionary fuzzer was developed that attempts to increase the code coverage of fuzzy input in hopes of unveiling bugs not found with more primitive forms of fuzzing. Three open source CoAP server applications were tested with varying degrees of success. The overall code coverage measurements and number of bugs encountered did not show enough progression to support the technique as an effective tool to use when fuzzing CoAP applications. Further research opportunities exist as this research only tested a subset of available evolutionary algorithms and little investigation had been made on the contributing factors for the technique's practical ineffectiveness.

Sammanfattning

Sakernas Internet (IoT) är ett begrepp med en bred användning som beskriver när fysiska objekt är uppkopplade till internet. Dessa apparater är ofta attraktiva mål för cyberattacker på grund av den stora mängd apparater som ofta utplaceras med likadan mjukvara. Testningen av sådan mjukvara är av stor vikt i arbetet mot datoriserat missbruk. Fuzzing är en testningsteknik där man utsätter mjukvara för en stor mängd indata för att orsaka en krash i hopp om att hitta sårbarheter innan illvilliga aktörer gör det. Constrained Application Protocol (CoAP) är ett nytt applikationsprotokoll som är specifikt designat för användning inom IoT-apparater vars implementationer kan innehålla sårbarheter. Att använda fuzzing på CoAP-mjukvara har gjorts i tidigare skede med framgång, men det finns utrymme för utveckling. I denna rapport utförs en utforskande studie inom möjligheterna att använ- da evolutionära algoritmer till att förstärka effektiviteten hos vanliga former av fuzzing på CoAP-mjukvara. För att göra detta utvecklades en så kallad evo- lutionär fuzzer som har som mål att öka täckningen av koden vid testad indata på CoAP-implementationer. Detta i hopp om att stöta på fler sårbarheter än vid mer primitiva versioner av fuzzing. Tre olika CoAP-applikationer med öppen källkod testades med den skapade fuzzern med olika grader av framgång. Kod- täckningens mätvärden och antalet sårbarheter som upptäcktes visade sig inte vara nog för att stöda effektiviteten av evolutionär fuzzing vid användning på CoAP-applikationer. Möjligheter för vidarestudier öppnades då denna studie endast testade en delmängd av tillgängliga evolutionära algoritmer och endast mindre undersökningar gjordes på anledningarna till teknikens praktiska inef- fektivitet.

Contents

Chapter 1

Introduction

It is likely that the expression *Internet of Things* (IOT) was first used in 1999 as part of a presentation on the potential for Radio Frequency Identification (RFID) to be used in a company's supply chain, mostly to attract the executive attention on the back of the then trending topic of internet [1]. It has since evolved from a sales pitch catch phrase to a worldwide phenomenon. Due to the recent advances in wireless communication it has grown into a widely used expression that encapsulates the concept of intertwining the physical world with the information world; where any object could be internet connected and talk to other machines on behalf of humans [13][25].

IoT environments are considered constrained when the devices in it have physical restrictions such as limited memory and battery power. To improve the performance of these constrained environments, special protocols and software are created with efforts to reduce overhead in network communication. One such protocol is the Constrained Application Protocol (CoAP). CoAP aims to bring the Representational State Transfer (REST) architecture in an appropriate form to constrained environments. It is a generic web protocol with similar characteristics to the Hypertext Transfer Protocol (HTTP) [23]. Due to the popularity of the REST architecture it is likely that CoAP will be widely adopted in the future.

With the growth of IoT comes new security issues along with an increase in severity of existing security issues. Big reasons for this is due to the diversity of the devices and the large scale it is often deployed in. Security flaws can be found in either the designs of individual devices or in the communication between them [32]. It is therefore important to review and test the security in both the actual devices and the protocols they use to communicate. Security flaws in devices can exist in both hardware and in software. The software implementations made to handle protocols such as CoAP could be vulnerable and therefore be an attractive target for malicious actors.

One way to test software applications for vulnerabilities is *fuzzing*, short for fuzzy testing, named from the way line noise from "fuzzy" telephone lines used to crash modem applications [20]. Fuzzing is the process of sending irregular data to a system with the purpose of crashing it, thus revealing reliability issues in the system [24]. It is an easily automated testing technique that covers a multitude of boundary cases that would be expensive using other comparable testing techniques. While it appears primitive in nature, fuzzing is surprisingly effective and has been responsible for uncovering severe vulnerabilities in, for example, software created by companies such as Microsoft [20].

A limitation of normal fuzzing is the significant role randomness has in its ability to find bugs. Certain functionality in software systems could depend on specifically structured input and could therefore be hard to reach using random data. Both static and dynamic analysis techniques can be used to increase the accuracy of input creation. One method to improve the efficiency is to use evolutionary algorithms in the fuzzing process. By monitoring the execution and calculating the *fitness* of each input, a new *generation* of inputs can be created by combining and mutating the most fit from the previous generation. The fitness is often measured using the amount of covered code using the intuition that executions with higher code coverage has increased probability of finding bugs [24]. Evolutionary fuzzing of different forms has been used to test software applications with reported success [7] [21] [29].

It will most likely be important to use fuzzing to test the emerging IoT software applications, as the number of IoT devices increases on the market. There has some been effort to employ fuzzing for IoT, but there is room for further research. CoAP stands out as a candidate for further testing due to the expectations of future usage. CoAP implementations has been fuzzed using primitive fuzzing techniques in the master's thesis by Melo [17] with great success but it is reported that further research in more advanced fuzzing methods on IoT could be useful. There is a possibility that evolutionary fuzzing is an effective method of testing the robustness of applications of CoAP and would therefore provide a tool to increase the reliability of IoT devices.

1.1 Research Question

It is not entirely obvious how evolutionary fuzzing will behave in the scenarios like the network protocol fuzzing of applications that implement protocols such as CoAP. An intuition is that the evolutionary algorithm would have trouble developing properly due to the homogeneous nature of the packets used in network communication, but it is also intuitive to expect the fuzzer to construct better inputs than its non-evolutionary counterparts.

The question this thesis ultimately aims to answer is

- Is robustness testing with fuzzing on CoAP applications improved with the employment of evolutionary algorithms?

Providing an answer to the question will require a explorative study on material relating to the CoAP specification, fuzzing, and evolutionary algorithms. The objective is to create an evolutionary fuzzing environment so the effects and performance can be properly monitored. The evolutionary fuzzer will be tested against available CoAP implementations to get measurements on real world applications. To achieve the goal of answering the research question, comparisons between normal fuzzing and evolutionary fuzzing and quantifications of how well the evolutionary algorithm proceeded in terms of covering code and causing crashes will be used as arguments for the technique's success.

1.1.1 Scope of Study

The study will only focus on fuzzy testing and will not be compared to other robustness testing methods. It will also be limited to only testing CoAP applications. An extensive root cause analysis for potential errors will not be performed in order to stay within the limits of the research question.

Chapter 2

Background

To sufficiently test implementations of a protocol we need to both understand the existing techniques of fuzzy testing and comprehend the structure of the protocol. This chapter provides the knowledge necessary to understand the problem that is examined in this degree project. Both a summary of the Constrained Application Protocol's specification and information about different aspects and techniques of fuzzing are presented. This information is used as a foundation for approaches used in the method and serves as a reference point in the discussion and conclusions.

2.1 Constrained Application Protocol

In this chapter we look at different aspects of CoAP to get familiarized with the protocol's internals and characteristics.

Constrained Application Protocol, abbreviated to CoAP, is a web protocol designed for the special requirements of resource-constrained computer environments, where devices have limited capacity in qualities such as processing power, storage/memory, and battery power. An example of a constrained environment is temperature monitoring where battery powered devices monitor could measure temperatures and communicate over wireless communication. An advantage in these environments is for the devices to consume as little energy as possible. One design goal with CoAP was to reduce message overhead to avoid network fragmentation of the packets that could otherwise cause a significant reduction in packet delivery probability. CoAP uses an interaction model that is similar to the client/server model used in the widely used Hypertext Transfer Protocol (HTTP). In this interaction model, requests are sent from a client to request an action using a method code on a server resource that is addressed using a Uniform Resource Identifier (URI), similar to HTTP requests. The server then performs the requested method action and returns an appropriate response to the client. The method codes form a subset of the Representational State Transfer (REST) format, and the defined methods are GET, POST, PUT, and DELETE. GET is used to access a representation of the information stored on a resource. The function of POST is determined by the server executing it but is usually used to create or update a resource. PUT updates the targeted resource and DELETE deletes a resource [23].

2.1.1 Messaging Model

To successfully communicate over CoAP we need an understanding of how messages are sent and received over the network. CoAP uses User Datagram Protocol UDP as its transport layer protocol and each CoAP packet is designed to fit within the payload of a single UDP packet. Both requests and responses uses the same 4-byte header that is explained in detail in section 2.1.4. Each message has a 16 bit Message ID that can be used to detect duplicates and provide optional reliability. The reliability is provided by setting the message as confirmable. Confirmable messages (CON) are retransmitted after a timeout period and with exponential back-off between retransmissions until an acknowledgement message (ACK) is received. When a request is unable to be processed, a reset message (RST) is returned instead of the acknowledgement [23].

2.1.2 Resource Discovery

Discovering the available devices and other resources is an important part for the communication between a client and server, especially when it is done machine to machine and no human is there to guide the process. To discover the resources available on a server, CoAP implementations should support the CoRE Link Format [23]. The CoRE Link Format defines a well known URI that is used to retrieve the resources that are hosted on the server. This is defined as the relative URI "/well-known/core" and delivers the resource links in the payload of the response [22]. Since all IoT devices running Coap defines this endpoint, it is well suited to be used to check the liveness of a service, and because it requires specialized logic in the protocol, it could also be tested for vulnerabilities.

2.1.3 Security Options

CoAP is designed with security in mind, and it is therefore important to view what built in features it provides and does not provide. CoAP provides no protocol primitives for use in authentication or authorization. There are a number of security options defined in the CoAP specification and they are based on the Datagram Transport Layer Security (DTLS).

NoSec No protocol level security which means that DTLS is disabled. Can only be secured by keeping attackers from being able to send and receive packets.

PreSharedKey DTLS is enabled and there is a list of pre-shared keys and each key contains a list of which nodes it can communicate with.

RawPublicKey DTLS is enabled and the device has an asymmetric key pair without a certificate. The device has an identity calculated with the public key and a list of nodes it can communicate with.

Certificate DTLS is enabled and the device has an asymmetric key pair with an X.509 certificate signed by a common trust root. The device also has a list of trusted roots that can verify certificates.

The three last options are indicated by the "coaps" scheme and a DTLS-secured CoAP default port[23].

2.1.4 Message Format

This section describes the format of a CoAP message. It is important to study this closely, as it forms the entire attack vector that is used to communicate with a server and is therefore directly related to way the fuzzer has to work.

Each CoAP message is designed to be compact and the goal of the design is to fit within one UDP packet. They are encoded in a simple binary format. The message starts with a fixed size 4-byte header. This is followed by a variable length token field, which can be between 0 and 8 bytes long. After the token comes a sequence of 0 or more options. The last part of a message is the optional payload separated from the options with hexadecimal 0xFF. Figure 2.1 shows a visual representation of a CoAP message's format.

Header

The header has a fixed size of 4 bytes. The first two bits are the CoAP version number (Ver). It must be set to 1 in the current version. The next two bits represents the type of the message (T). These can be Confirmable, Non-Confirmable, Acknowledgement, or Reset with values 0, 1, 2, and 3 respec-

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Ver| T |  TKL  |      Code     |          Message ID           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Token(if any, TKL bytes) ...                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Options (if any) ...                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 1 1 1 1 1 1 1|Payload(if any) ...                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.1: CoAP Message Format - `https://tools.ietf.org/html/rfc7252`

tively. The four bits after the type is the token length (TKL), and represents the token length in bytes. Only values 0-8 are allowed here because 9-15 is reserved. Then comes an 8 bit unsigned integer that represents the code of the message. The code field is split into two parts called class and detail. Class is the three most significant bits and there are four different values this can have that are not reserved:

- request (0)
- success response (2)
- client error response (4)
- server error response (5)

The detail part is the 5 least significant bits of the code and combined with the class it specifies the method of code. An example: the class is 0 and the detail is 1 (0000 0001) means the message is a GET request. More information is found in RFC7252 in sections 12.1.1 and 12.1.2 [23]. The last 16 bits of a CoAP header is the message ID. It is used to detect message duplication and to match Acknowledgement/Reset messages with Confirmable/Non-confirmable messages. Message ID must be generated by the sender of the Confirmable/Non-confirmable message and must be echoed by the receiver in the Acknowledgement/Reset message [23].

Token

Tokens have values between 0 and 8 bytes and correlate requests with responses. They are generated by the client and intended to provide client-local identifiers of concurrent requests [23]. Separate message ID and token fields exist and are needed for the receiver to distinguish retransmissions from new messages.

Options

The option fields specifies a number of options in a message. Each option in the field is specified using an option number of the defined option, the length of the option value, and the option value itself. Option numbers can represent values like Uri-Path and Content-Format, they are specified in section 12.2 of RFC7252 [23]. Options must appear in the order of the options numbers, and they are separated by specifying the difference (delta) in option number for the current option from the previous option value. An option number is calculated using its delta plus the previous option number, with the first option using 0 as the previous option's number. An example of this is if the first option number $Option[1]$ is 2 and the second option number $Option[2]$ is 5, then the first option delta is encoded as 2 because $Option[1] = Option[0] + \Delta = 0 + 2 = 2$ and the second option delta is encoded as 3 because $Option[2] = Option[1] + \Delta = 2 + 3 = 5$. The third option would be calculated as $Option[3] = Option[2] + \Delta = 5 + \Delta$. Multiple options with same number can be used by setting delta value to 0.

An option format consists normally of a 4-bit Option Delta followed by a 4-bit Option Length and then the Option Value. There are special cases when either the Option Delta or Option Length has value 13 or 14 that adds options for bigger values. With 13, any or both values add an 8-bit unsigned integer each before the Option Value that specifies either delta or length minus 13. Value 14 adds a 16-bit integer in network byte order that specifies the delta or length minus 269. Value 15 is not allowed in either the delta or length as it is reserved for the payload marker (separator between options and payload). Figure 2.2 shows the structure of an option. The option value size is exactly the number specified in the option length [23].

Option values can have different formats. An *empty* value is a zero-length sequence of bytes. An *opaque* is an opaque (unspecified format) sequence of bytes. *Uint* is a non-negative integer represented in network byte order using the number of bytes specified in the option length field. It is allowed to have integers with leading zeroes. A *String* value is a Unicode string that is encoded using UTF-8 in Net-Unicode format [23].

```
  0   1   2   3   4   5   6   7
+---------------+---------------+
|               |               |
|  Option Delta | Option Length |   1 byte
|               |               |
+---------------+---------------+
\                               \
/         Option Delta          /   0-2 bytes
\          (extended)           \
+-------------------------------+
\                               \
/         Option Length         /   0-2 bytes
\          (extended)           \
+-------------------------------+
\                               \
/                               /
\                               \
/         Option Value          /   0 or more bytes
\                               \
/                               /
\                               \
+-------------------------------+
```

Figure 2.2: CoAP Option Format - `https://tools.ietf.org/html/rfc7252`

Payload

The payload is an optional field. If it is of non-zero length it is prefixed by a fixed 1-byte payload marker with hexadecimal value of 0xFF. It extends from the end of the marker to the end of the UDP packet. An absence of payload marker means a zero length payload. A payload marker follow by a zero length payload should be processed as a message format error.

2.2 Fuzzing

***Fuzzing** (fuzzy testing), a software testing technique, should not be confused with **fuzzy logic**, a form of logic handling the concept of partial truth. There is no connection between the two areas of study, except for the name.*

Fuzzing is a highly automated software testing technique that works by sending specially crafted input to applications with the intention of uncovering bugs. The purpose of fuzzing is to better ensure the absence of exploitable vulnerabilities by covering boundary cases in the application's logic. It allows for testing of large amounts of test cases that would otherwise be too costly to do by hand. When creating software with an external surface that takes input from untrusted sources, it is important to test boundary cases and a fuzzing software could make this process significantly easier [20].

The first credited source of fuzzing is from Miller, Fredriksen, and So [18] in 1990 when Unix utilities crashed when exposed to accidental random input. Further testing ensued and it was discovered that fuzzing was a surprisingly effective method of uncovering fatal bugs in programs [18].

Software and frameworks that perform fuzzing are usually called fuzzers and since the 1990s it has transformed from a niche technique to a full testing discipline. Even though the world of fuzzing has changed it still has one true goal; to crash the system under test by stimulating a multitude of inputs to find any reliability and robustness flaws. Fuzzing is a pro-active testing approach that can be used without previous knowledge of vulnerabilities and is therefore good at finding new vulnerabilities. It can be used to test the security of any process, service, device, system, or network without requirements of fully knowing the supported interface [24].

2.2.1 Black-box Fuzzing

It is called black-box testing when a testing technique is performed without assumption of any prior implementation specific knowledge. Only the external specifications of the application is assumed to be known [14]. A black-box fuzzing technique is employed when a fuzzer only utilizes the external interface to craft fuzzy input and relies on the responses from the system under test [24].

This fuzzing technique gained traction in black-hat communities when it was used to search for vulnerabilities in commercial software products. One of the reasons for its uptake was that it did not need access to any source code [15]. A major disadvantage with black-box techniques is that they never monitor the application's inner state and is therefore likely to only explore and test a small fraction of the tested application [8].

Example of black-box inefficiency

A commonly used example to show when black-box testing is inefficient is if we encounter code that looks like:

```
if( x == 10 ) then {
    doSomething();
}
```

and x is a 32-bit integer affected by the input. A black-box testing system will have no idea about whether or not we entered the *then*-statement. And if nothing is known about the internals of the system it will be a probability of $\frac{1}{2^{32}}$ of actually entering it. If that part of the code happens to be vulnerable, we have a low chance of exposing any of its vulnerabilities.

2.2.2 White-box Fuzzing

Different from black-box testing but similar to many other software testing techniques, white-box testing is based on the assumption of internal knowledge of the tested application. The adequacy of test cases generated with white-box techniques is assessed using metrics observed in the systems. This metric is usually coverage of either the system or another model representing the system [26]. A popular coverage metric is code coverage and is measured by observing the control flow of the program during test cases. Typical control flow metrics are statement, branch and path coverage; and the intention is to cover as many of them as possible [2] .

Fuzzing using the white-box testing technique is used to counter the inefficiency of black-box techniques exemplified in 2.2.1. Because it is a white-box scenario the tester can observe the source code and retrieve measurements from the execution of tests. In the work by Godefroid, Levin, Molnar, et al.

[11] a white-box fuzzer was invented that symbolically executes the program with fixed input values and gathers input constraints from conditional statements. A constraint solver is then used to create new inputs that execute different paths in the program [11]. This fuzzer called SAGE has been used in practice and has had a remarkable impact in finding bugs in Microsoft applications. Sage was credited for finding a third of all file-fuzzing bugs during the development of Windows 7. It is even more impressive considering it was usually run last in the development process, indicating these bugs were missed by every other software testing technique [10].

There are problems with white-box fuzzing that should be considered. It suffers performance issues with large code bases, as it causes severe overhead in comparisons with black-box testing. A problem is also that it can focus on the wrong things such as code coverage when it does not necessarily result in a better bug finding capability [11].

2.2.3 Grey-box Fuzzing

White-box infers knowledge of source code in the application and black-box refers to tests run on already compiled code without knowing the structure of the application. Combining these could create new opportunities to improve the achieved results. This hybrid technique is called grey-box [24]. The definition for grey-box testing seems to vary in different reports and articles, but the resulting core aspects of the grey-box testing methods are similar.

In the paper by Tyler and Soundarajan [27] a grey-box monitor technique was used to get more information from the testing process. They added an auxiliary *trace variable* [5] to the application to save the states and to hook method calls performed within a specifically selected method [27]. As they did not use or alter any source code it can not truly be classified as a white-box method but information about the implementation is gathered so it also rejects the classification as black-box. This is an example of how grey-box techniques can be used to gain more information without strict requirements on the system.

Applying the grey-box techniques to improve fuzzers has been done and one famous grey-box fuzzer is called American Fuzzy Lop (AFL). It uses compile time instructions and genetic algorithms to automatically create test cases. AFL has been used to find some very notable bugs in systems such as smart phone platform Android and open source DNS server BIND [12].

2.2.4 Evolutionary Fuzzing

Evolutionary testing is a software testing technique that uses ideas and concepts found in the study of genetic algorithms [24]. Genetic algorithms use combinations of previous inputs to optimize the effectiveness of the next input. This is done by culling the inputs with bad performance while mixing those with good performance with each other to form the next generation of inputs. The idea is loosely based off genetics and is often compared to Darwin's theory of evolution. For these algorithms to apply to software problems a numerical fitness value must be computable from an execution to allow for the comparing of input performance [31]. As with white-box testing, code coverage is a possible candidate for this fitness value. Using code coverage as fitness metric, a fuzzer would let the input that caused high code coverage to survive while discarding the inputs that yielded low code coverage. Evolutionary testing can be considered as either white-box or grey-box testing depending on what is required to measure fitness in the application. It can be classified as grey-box because it is possible to calculate code coverage without access to source code using a debugger attached to the application [7].

Evolutionary algorithms has been used with good effect on fuzzing and one example is the Evolutionary Fuzzing System (EFS). It utilizes evolutionary algorithms to both learn the targeted protocol and to find effective fuzzing input. The goal of the system is to reduce the developer times while providing a better probability of finding bugs. EFL uses code coverage as its fitness metric [7].

Code coverage as testing metric has been criticized for not always providing an accurate representation of the performance of a specific input. More specifically, high code coverage does not imply a better bug finding capability [11]. The reasoning is that even if 100% of the code is covered, some vulnerable parts could have been executed with benign data and not triggered bugs that may exist. In response, it is then argued that the metric is still useful because it is realistic to assume that less tested code has a higher probability to contain trivially triggered errors [4]. Another argument with similar reasoning is that no bugs are found in code that is not executed [24]. However, one can see that two branches of execution may be exclusive and one of the two provides better code coverage than the other. An evolutionary fuzzer would conventionally choose the branch with higher code coverage and the other branch is never tested. If there exist vulnerabilities in the lower code coverage part, the fuzzer would miss them. To counter this, Rawat et al. [21] applied weights on blocks of code to indicate the importance of reaching that part. Less frequently reached code would generally get a higher weight and easily reached code would get a lower weight. An execution of a testing input would accumulate weight based on what code it executed and it would be given a score directly calculated from the weights. This would then be used as their fitness value [21].

2.2.5 Fuzzing Test Case Generation

Fuzzing has grown in popularity because it is easier to generate inputs automatically than to manually create them by hand. Both quality and quantity of the fuzzing inputs are two of the most important aspects of successfully finding vulnerabilities [19]. This section describes some categories of fuzzing input construction and what the advantages and disadvantages of them are. The art of creating fuzzy input is often only referred to as *fuzzing*.

Random Fuzzing

Random fuzzing is a simple method of creating fuzzing input. It works by using an algorithm that generates data at random and sending it to the application. It has potential to detect a lot of bugs but the inputs will often be considered invalid and rejected by the application before any vulnerable code has been reached. [28].

Generation-based Fuzzing

Generation-based fuzzing is a test case generating technique that involves building entire inputs from the ground up. This is not to be confused with the word *generations* that might appear when describing iterations in evolutionary algorithms. To perform generation-based fuzzing, a specification of the application is used to describe how the inputs should be constructed. This specification could be an RFC or something similar. The key to making effective test cases is to make them different from valid data but at the same time make them similar in structure. Valid data would probably pass through the application without errors and significantly different data would be rejected quickly. The advantage of using generation-based fuzzing is that it is very effective at finding bugs due to the easy generation of semi-valid fuzzy input once the specification is understood. The drawback is that it takes a lot of upfront work to understand the specification [19].

Mutation-based Fuzzing

Instead of constructing original input for the tested application, one can use the technique called mutation-based fuzzing. This technique avoids the challenge of understanding the specification by using existing valid input as its seed. It then mutates the valid input by performing actions such as random bit flipping to cause problem in the application. More advanced methods of selecting mutations that uses mathematical models for optimization of the mutations to increase its bug finding capability have been proposed [6]. The benefit of mutation-based fuzzing is that there is no requirement of knowledge about the application beforehand and is therefore easy to create. A disadvantage is that there are not many possibilities for different inputs as the fuzzer has little knowledge about the structure of the input which could lead to lower bug finding capability [19].

Intelligent and Dumb Fuzzing

The names intelligent and dumb (non-intelligent) fuzzing appears in a lot of texts about fuzzing and they refer to the notion of whether or not a fuzzer knows the protocol (specification). Intelligent fuzzing has information about the specification while the dumb fuzzer does not. It is in general expected that intelligent fuzzing covers more code than the dumb and thus uncovers more bugs. Intelligent fuzzers are, however, more time consuming to create [24]. A generation-based fuzzer is obviously intelligent (otherwise it would be equivalent to a random fuzzer) but a mutation-based fuzzer could also be intelligent if it made intelligent mutations such as repeating certain fields in protocols.

Chapter 3

Related Work

This chapter provides a more detailed view of related work that directly relates to this project.

3.1 CoAP Fuzzing

Fuzzing on CoAP implementations is not thoroughly researched and the only publicly available work that could be found at the time of writing this was the publication by Melo, Geus, and Grégio [16] and its corresponding master's thesis by Melo [17]. This section will mostly present information from the master's thesis as it is the more thorough presentation and both the master's thesis and the publication are based on the experiment. The study is focused on the use of black-box fuzzing to test the robustness of currently available CoAP server implementations. A fuzzing framework is presented that utilizes five different black-box fuzzing techniques: random, informed random, generation-based, mutation-based, and smart mutation-based fuzzing. The regular random fuzzer would randomize entire CoAP messages (random payload in the UDP packet). To give the random fuzzer a fair chance they constructed another version called the informed random fuzzer that only randomizes selected parts of the CoAP message to avoid the immediate rejection of the input. The smart mutation-based fuzzer is a more intelligent version of the mutation-based fuzzer that mutates specific parts of a CoAP message instead of performing simple bit flips at random. These mutations follows certain mutation-rules such as testing 0, 1, and -1 when an integer is expected. They were adapted from the rules made in the work by Vieira, Laranjeiro, and Madeira [30] to make them suitable for CoAP fuzzing [17].

The implementations they chose to perform fuzzing on were searched for on the internet using a number of search engines and repositories. Because of CoAP's recent conception there are not a lot of implementations out there, and only a few of commercial products use the protocol. They chose any application they could find that listened for CoAP packets on a UDP port [17].

Important concepts of false positives and reproducibility are brought up in the study. Not all errors found with a fuzzer are guaranteed to be actual flaws in the system. Sometimes it could be something that went wrong with the heartbeat pulses such as timing out before the application was able to respond. Not all errors found with a fuzzer are guaranteed to be actual flaws in the system, and some may be hard to find and reproduce. A majority of the non-reproducible errors were found in implementations built in C and it is assumed that it is because low level languages are more prone to race conditions and other issues that might affect robustness and reliability [17].

The results of the study were impressive. A total of 100 errors were found in 14 out of 25 applications. There was a 83% error reproducibility. The average false positive rate was 31,3% but in implementations that found at least one true positive error (a real error), the false positive rate was 1,86%. A conclusion of the study states that different or combinations of fuzzing techniques that cover more code could be used to find more vulnerabilities [17].

This master's thesis separates from Melo's master's thesis in that this one will explore the effect of evolutionary elements in fuzzing of CoAP applications while Melo's only examines the effect of black-box fuzzing. Developing an Evolutionary Fuzzer is more a complex and time-consuming process than for a black-box fuzzer which means that this report will not be able test as many applications as Melo.

3.2 Evolutionary Fuzzing

There are many papers on the subject of evolutionary fuzzing and they have different applications and use different methods for their evolutionary algorithms.

VUzzer is an advanced evolutionary fuzzer that uses Dynamic Taint Analysis (DTA) to evolve its input. DTA works by tainting input and during execution examine which operations are affected by the tainted input. DTA in VUzzer is mainly used to satisfy conditional statements with strict requirements to cover more obscure code paths and to avoid code that handles errors which, if visited, would indicate rejection of input. For fitness calculation they use weighted blocks of code to give paths a distinct fitness value, where less frequently visited code is given a higher weight and therefore higher priority for the evolving input. After successful results they conclude that their method is a viable and scalable strategy [21].

IFuzzer is another fuzzing tool that uses evolutionary algorithms to improve itself. It targets JavaScript interpreters and was used on an older version of Mozilla's JavaScript Interpreter with success. To use fuzzy testing on interpreters, code is generated and sent to the interpreter with the hopes of crashing it. Unlike other evolutionary fuzzers, IFuzzer does not monitor the targeted application for code coverage. It instead just listens for crashes, warnings, and errors etc. It also analyzes its own generated JavaScript code statically using structural metrics such as cyclomatic complexity to generate as unusual code as possible. They conclude that the technique works but there exists room for improvement, especially in the rules of mutation between generations [29].

Evolutionary Fuzzing System (EFS) is one of the first fuzzing projects to feature evolutionary algorithms to improve testing input and is presented in both the publication by DeMott, Enbody, and Punch [7] and in the book by Takanen, Demott, and Miller [24]. EFS was presented as a revolutionary grey-box fuzzer that could be applied to any interface without needing to know any details about its specification. It instead learned the protocol using genetic algorithms, and in the same time creates better fuzzy input each iteration. A framework called PaiMei was used to pre-analyze binaries, attach debugging flags, and then measure code coverage for each test. The evolutionary algorithm was using a layered mutation approach where sequentially tested input was grouped into full transactions called sessions, and sessions are group together into pools. Fitness was maintained in both a session level and a pool level. Sessions are used because an application may have persistence between inputs that could cause different behaviour and pools are used because groups of less fit sessions could be better when measured as a group than any single fit session, which effectively provides a solution to evolutionary algorithms only selecting a long path instead of a potentially vulnerable shorter path. Maintaining fitness on a session level was done by both crossing over sessions so that the inputs themselves used are mixed among each other and by mutating the inputs according to a set of mutation-rules. The fitness metric measure in sessions was the coverage of each session. Maintaining fitness on a pool level was also done with crossovers and single pool mutations. Pool crossovers are similar to session crossovers; sessions are mixed together from different pools but the fitness was calculated using the sum of unique code covered for all sessions in the pool. Mutating pools was done by removing sessions and adding new sessions according to session initialization rules. EFS was used to test a software application called Golden FTP server with success and it was concluded that the two-layered evolutionary process with sessions and pools was effective [7].

Chapter 4

Methods

To provide an answer for the question this report aims to answer, an experimental suite was developed that is able to start a CoAP server, send a set of testing parameters in forms of CoAP packets, measure fitness of the running server, and use the results in an evolutionary algorithm to provide a new set of testing parameters. By recording the code coverage of each test set and record the crashes, a projection of the success of using evolutionary algorithms in CoAP fuzzing will be provided.

This chapter will explain in detail the practical experiments that were done to provide this report with quantifiable results. The content will serve as a foundation for the discussion and ultimately the conclusion. As large portions of the topics of fuzzy testing and evolutionary algorithms involve intuitive thinking and often has a stochastic outcome, the decisions made for the experimental suite will be accompanied with a thought process or argument for the choice of method.

4.1 Targeted Software

In order to get as accurate results as possible, available open source software of CoAP server applications were used as targets for the fuzzy testing. There are both good and bad consequences of choosing this method instead of using a specially prepared software as target for the testing,. An already available open source system reflects the real world directly and any vulnerabilities discovered would irrefutably favor the testing method. Another perk is the time saved by not having to develop a custom CoAP server application. The issues that arise with using existing software are that there are no certainties of the existence of vulnerabilities and the potential lack of support for diagnostic tools. We

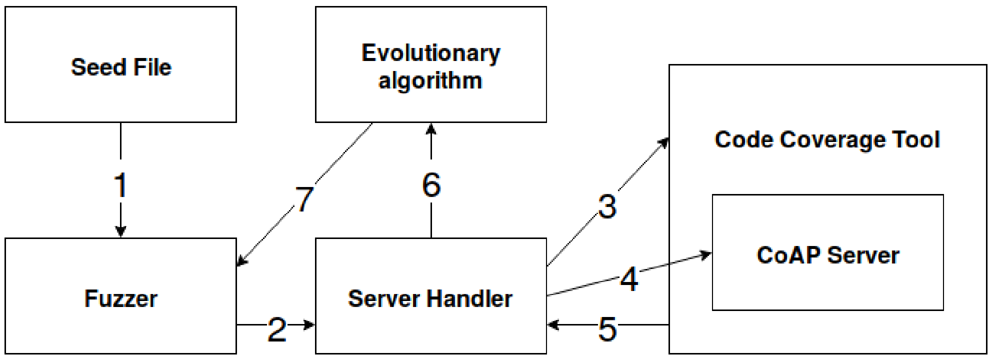

Figure 4.1: Flow diagram of test suite

could for example systematically measure the effect of testing methods if the software had known planted vulnerabilities.

Due to time constraints and the desire to research with measurements based on the real world, the decision was made to use already available CoAP server applications. In the work by Melo [17] all available CoAP servers were fuzzy tested but because this experiment's method has requirements such as compatibility with fitness measuring and increased overhead due to process monitoring and fitness calculation, only a set of selected server applications will be tested. Some servers may for example require a different operating system and would require lots of work to make compatible with the experimental suite. The selected servers are listed in the results.

The server binaries used in the tests are the example or test servers provided in the base source code that allow the user to test the software without initial configuration. These were chosen to reduce the time required to set up the experiments, as they are already working server applications. If more than one example binary exists, the one exposing the most endpoints will be chosen.

4.2 Fuzzing Software Suite

This section provides an overview of the system developed for this experiment. A flow diagram of the important parts of the fuzzing suite is seen in Figure 4.1.

To start the fuzzer we include a seed file that contains a simplified form of CoAP message. This includes small packets containing a method, URI, and payload if necessary. In Figure 4.1 in action 1, the fuzzer parses these seed packets and places the information of them into the internal CoAP message structure. The fuzzer uses these seed packets to generate an initial set of test parameters. These parameters are in the form of CoAP messages. Action 2 is the process of the fuzzer telling the server handler to send these inputs to the targeted server. The server handler then starts the server using the Code Coverage Tool as seen in action 3. The CoAP messages are sent to the server in action 4. After the CoAP server has finished the processing of all the messages the server handler sends a final input that, if the server is running, will always yield a response from the server, to check if the server has crashed. The server handler then shuts down the server to let the Code Coverage Tool write the code coverage information. At action 5 the Server Handler reads the code coverage information from the Code Coverage Tool and translates it into a fitness value. The fitness values are sent to the Evolutionary Algorithm in action 6. The Evolutionary Algorithm analyzes the fitness values and decides what mutations to make and what parameters to cull in order to create a more fit next generation. This decision is sent to the Fuzzer in action 7 and the whole process repeats itself in action 2.

To switch between the targeted CoAP server applications the only thing that in theory should need is to change the CoAP server module, and monitor the other application with the Server Handler and the Code Coverage Tool. But in the real world there may be discrepancies in how the server is run and what endpoints are available, and manual adjustments may be required. The endpoints (that may differ between server applications) can be tested by simply changing values in the seed file to that the fuzzer bases the parameters on the available endpoints, in the attempt of triggering more server functionality. A more vicious problem is a that servers take different time to start and some servers may spawn additional processes, making it harder for the Server Handler to predict which processes to end. This requires hands on testing to make sure the system works the way it should on each server. For the Server Handler to be robust enough to reliably handle a server it may need to delay its operations to allow the server time to act. This causes delays in the system, causing it to be slowed down. If it fails to wait for the CoAP server we could experience false positive results in crashes. To not report any false positives, all packets that causes errors are tested again for reassurance.

4.3 Fitness Calculation

To use an evolutionary algorithm we need to calculate a fitness value for each parameter we are testing, to decide on which parameters are more likely to cause crashes. Code coverage is used as the fitness value for this experiment. This is because a higher code coverage of a test is more likely to encounter bugs than a test with less code coverage.

Calculating code coverage for this experiment requires measurements based on the packets being handled in the server. There are several possible ways of performing this kind of monitoring. The best and most accurate one would be to start measuring code coverage once the server has started and then send one set of packets. After the set of packets has been sent, the code that was executed when handling the packets is counted and we start measuring again to count the coverage of the next set of packets. Unfortunately, no tool was found that properly supported this functionality resulting in a less convenient process to be used. This process is to measure code coverage of a server process from start to end, and reporting code coverage for the whole process. This will result in a lot of covered code that is related only to start up the server and the precision of packet based code coverage is lost. But because the difference in code coverage between packets is the same regardless of counting start up code coverage, it sufficed as the method to use.

The code coverage tool used in the experiment is an open source run time code instrumentation tool platform called *DynamoRIO* [9]. A major benefit of using DynamoRIO is that it does not require instrumentation before compilation, and can therefore be attached to pre-compiled binaries. It has a built in tool called *drcov* that is used to calculate code coverage. To run this you first run DynamoRIO with the tool drcov attached and feed it with a shell command that runs the server binary, and once the server binary process is finished or killed, DynamoRIO writes the code coverage information in a log file. The code coverage log contains the coverage of the code in terms of basic blocks. Because the log not only counts the code of the binary, but also the linked libraries, the decision was made to filter out only the modules that belong to the source code of the server application, and discard the coverage of the linked libraries. The reason for this is that the purpose for this project is to test the robustness of the actual servers, and instrumenting the evolutionary algorithm with information that belongs to external code could steer it of its intended purpose of covering the source code. Code coverage is parsed in the form of an absolute value rather than a percentage based value because DynamoRIO only reports the basic blocks that have been covered. This is fine for coverage comparisons but results in difficulties knowing how much of the code base has not yet been covered, which could be useful in strengthening conclusions made on the completeness of the tested parameters.

4.4 Crash Monitoring

The purpose of robustness testing is to make the application crash in an unintended way, so that it can be fixed before any consumer encounters it. This means that there needs to be functionality in the test suite to check if the application has crashed. Server applications can be considered alive if it responds to requests made to it. As a result, it could emit false negatives to check if the process is running as a liveness check. The false negatives could come if the server process is running but it cannot reply to requests due to specific state or bug in the application. A well known request is therefore used to check the liveness, as it will yield a reply if the server is alive and timeout if the server considered dead. The request used is the /.well-known/core URI that most CoAP servers support and will always reply to.

Checking liveness occurs at the final stage of each set of test packets sent to a server process. To track the packets that caused the server to crash, the suite logs all the sent packets in a unique file once a crash has occurred, for later review. Due to the fitness tool used, the test suite will calculate the liveness request as part of the code coverage. This should not have an impact on the performance of either the fuzzer or the evolutionary algorithm because all set of test parameters will include it, and it is highly unlikely that a valid well known packet itself would cause any crashes.

4.5 Evolutionary Algorithm

Choosing the best evolutionary algorithm to use in fuzzy testing CoAP applications is a hard task that require careful research on the specifics of both CoAP as a protocol and the properties of algorithm. The focus of this research was to examine the effects of evolutionary fuzzing on CoAP applications and a decision was made to not put time and effort into researching the best evolutionary algorithms. Because the primary inspiration for this research came from the initial work from DeMott, Enbody, and Punch [7], the chosen algorithm became the two layer approach described in that paper. In this section the two layer evolutionary algorithm will be explained along with how was tailored to fit the CoAP protocol.

4.5.1 Components

Originally, the evolutionary algorithm described in DeMott, Enbody, and Punch [7] contained four data components named pools, sessions, legs, and tokens.

This made sense as they were fuzzing an FTP server called Golden FTP server. Tokens were pieces of data that could be of different types such as integer, character string, or binary string. A leg was a collection of tokens and performs either a read or a write with the tokens in its collection. A session was a collection of legs and represented a whole transaction with the target application. Sessions were grouped together to form pools. Fitness were calculated and maintained on a session and pool level separately [7].

Adopting the original components to work on a CoAP server is not complicated but since an FTP server and a CoAP server handle different types of inputs it is still a necessary process. A *CoAP message* is the primary component used in communications with a CoAP server. This contains all the information the CoAP server needs. The contents of each CoAP message will be subject to mutation but the evolutionary algorithm does not measure the performance of each message. Instead a *session* similar to the one used originally is used for the minimum single fitness value. A session in CoAP will comprise of a set number of CoAP messages. A session's performance is measured by starting the server and sending each contained CoAP message sequentially to the server before closing and calculating the code covered by all CoAP messages together as fitness. Groups of a set number of sessions are divided into pools that represents the outer layer of the algorithm. Fitness is calculated for each pool along with the fitness of each session inside the pool. Instead of adding each pool's sessions' fitness into a large sum, the sum of all unique code covered is calculated for all the sessions in the pool. The unique code covered by a pool is guaranteed to exceed or be equal to the fitness for any single session, because the worst case is that the single best session's code coverage is the superset of all other code coverage in the pool. Figure 4.2 shows an example of a pool containing two sessions where each session contains five CoAP messages.

4.5.2 Generating Seed Inputs

The initial inputs are generated through the use of a seed file that defines some basic properties of CoAP messages. These properties could for example be the URI, method and value of a request. Once loaded into the program they become a vector of example messages that is used for generating new input. At the start of an execution all the pools are filled with sessions that are filled with generated CoAP messages. An initial mutation is made on the generated messages to avoid sending valid packets to the server.

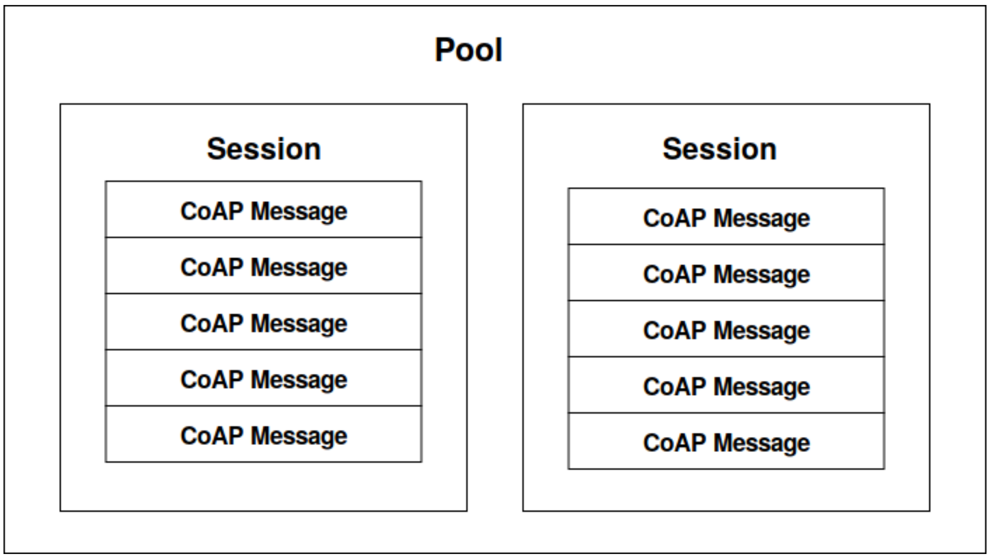

Figure 4.2: Example of a pool containing two sessions, each containing five CoAP messages

4.5.3 Evolutionary Mutation Operators

In the original version of the algorithm there were four operators performed on the components each generation. For pools there were Pool Crossover and Pool Mutation, for sessions there were Session Crossover and Session Mutation. The crossover operations mixes contents between two sessions or two pools by combining the first x elements of one operand with the remaining y elements of the other operand, forming a new session or pool. The mutation operations perform operand specific mutations on existing sessions or pools. Selection of operands depend on the measured fitness. Adapting these operations for CoAP was done in the following manner:

1. **Session Crossover** is done by first ordering the sessions of a pool according to their fitness value, and denote this generation as G. The most fit session is automatically copied into the next generation that we can call *G'*, ensuring the survival of the most fit. A selection of two random distinct sessions from G are mixed together to form a new session. To increase the survival of the fit there is a 70% chance of selecting a random session from the top half of G, and a 30% chance of selecting a random session from the whole of G. The mixing is done by using a random crossover point which is some index that exists in both sessions, and then connect the top part of the first chosen session to the bottom

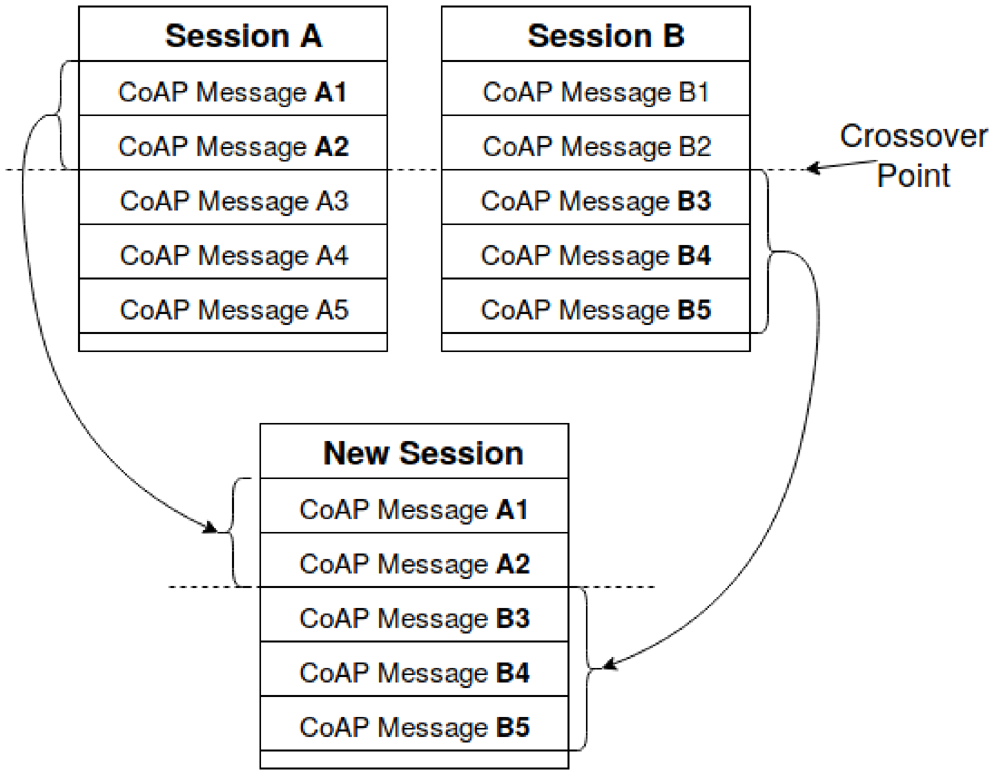

Figure 4.3: Session Crossover operation with crossover point 2

part of the second chosen session, where the top and bottom parts of the sessions is separated by the crossover point. An illustration of this operation is seen in Figure 4.3. The new session is then added to new generation G'. This session crossover is repeated until generation G' has the desired amount of elements.

2. **Session Mutation** is the act of modifying parts of the session by applying one of the mutation rules seen in section 4.6 on a random part of a CoAP message. In order to maintain the fitness, the most fit session of each pool is not mutated. Any other sessions has a set probability of being mutated.

3. **Pool Crossover** is very similar to *Session Crossover* but the fitness is calculated using a sum of unique code covered of all contained sessions and the crossover operation is applied to whole sessions instead of messages. Figure 4.4 depicts a pool crossover operation.

4. **Pool Mutation** differs from a *session mutation*. The most fit pool is

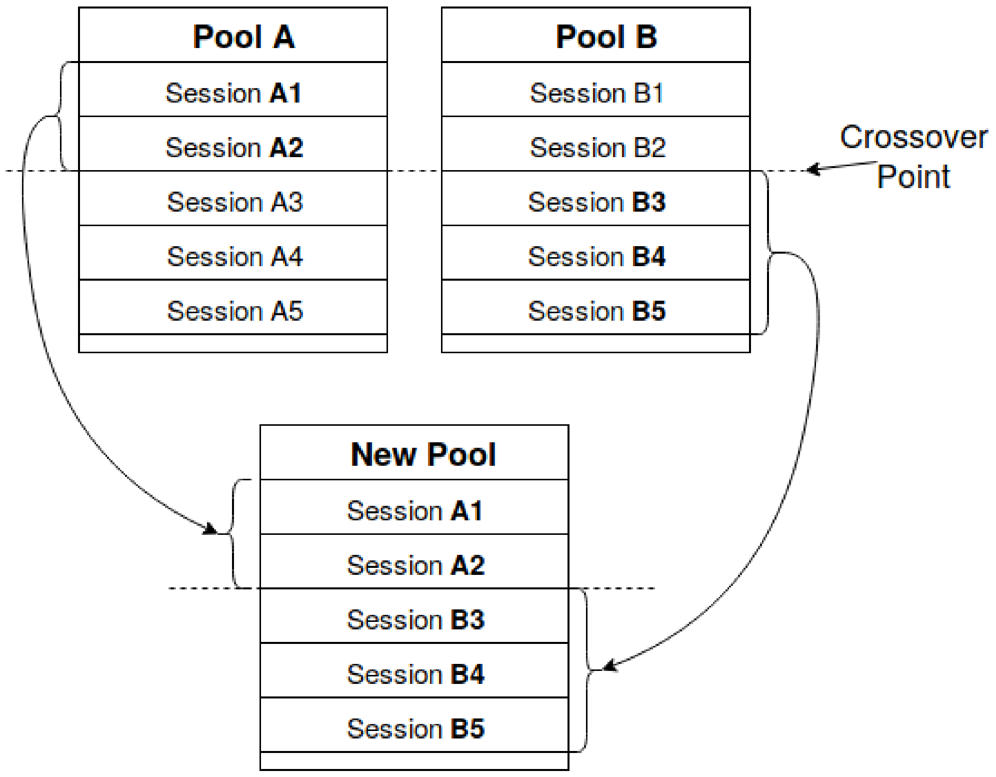

Figure 4.4: Pool Crossover operation with crossover point 2

again not mutated. All other pools have a 50% chance of adding a session with the help of the seed generation, and 50% chance of removing an existing session. If the size of pools have a set value then there is instead a 50% chance of both removing an existing session and adding a newly generated seed session.

4.5.4 Miscellaneous Evolutionary Algorithm Details

The original version of this algorithm used a fixed generation based wait before applying operations. Session crossover was applied after every generation, session mutation every seven generations, pool crossover every five generations and pool mutation every nine generations. For this experiment the interval for session mutations to occur is reduced to every three generations because the intuition is that it is more likely that malformed values in the packets will cause problems for the parser, rather than packets triggering an erroneous state in the server that would cause problems for future packets. The reason for this is because each message sent independently to the server and is parsed independently. Both individual parsing errors and combination errors are however possible so no changes were made on the timings of the other operations. During a session mutation, two random CoAP messages in the session are chosen as targets to perform mutations on, as the original algorithm would perform two different mutations on two different input tokens.

4.6 Mutations

The list of mutations to make on CoAP messages is an important building block in the fuzzer as they reveal all possible fuzzy input that can be created. Because mutations can be applied to already mutated messages, the combinations of mutations are important to consider when analyzing what input can be created. The mutation rules used in these experiments are heavily based on rules created for the smart mutational fuzzer used in the work by Melo [17]. These rules were used because they are tailored for the CoAP message format and has already been used to fuzzy test CoAP applications with success. An issue with this is that many of the inputs this paper's fuzzer creates will be similar to already tested inputs and may uncover fewer bugs in the earlier generations as they could have been fixed. A veiled benefit of testing already tested software with a new technique is that if the testing is successful, there is reason to believe the new technique is advantageous.

A table of the adapted mutation rules for different types and parts of CoAP messages are seen in Table 4.1. The types defined in the table are based of the types defined in the CoAP RFC [23], except for Payload which is a section of a message that can have different types between messages. In addition to the type-based mutations seen in the table, a bit flip mutation is also used. A bit flip changes the value of one random bit in the CoAP message. The targets for the mutations, both type-based and bit flip, are chosen among a number of sections defined in each CoAP message. Figure 2.1 shows the available sections that are individually mutated in the mutation engine, except for the Code section that is split into code class and code detail. These target sections have a different probability of being selected for mutations. The probabilities were chosen by the author to more often mutate the sections with a high number of possible values, to minimize the risk of testing the same value in a redundant manner. For example, the Version section of a message is currently required to be 1 according to the specification, and if it is not 1 then it should be rejected by the server. There is therefore little reason to mutate that part of a message as many times as we would change the options where a great number of values and combinations are valid. Each target section is given a set amount of tickets and when the mutation happens a random ticket is drawn to choose what target to mutate. This means that a higher ticket count means a higher probability of being mutated. Table 4.2 shows the ticket distribution used in this experiment.

The Option section of a message has special handling for its mutation due to its special structure. Options differ from normal targets because it can have different number of elements and uses special ordering and delta values. In case there is a mutation targeted for the Option section and there is no option available then a new option is automatically created and added. There is otherwise also a possibility for removing or adding an option. If options exist and no removal or addition of options occur, a random option is targeted for mutation. The option is mutated based on what type the option has and uses the same rules as for any other section of the message. Only the value of the option is used in the mutation, but the length fields of each option are also adjusted to fit the new value.

Type	**Test Name**	**Parameter Mutation**
String	StrEmpty	Replace by empty value
	StrPredefined	Replace by predefined string with random length in the range of the valid minimum and valid maximum, containing a random mixture of characters: '\0' 'a' 'A' '.' '\' '%'
	StrAddNonPrintable	Add non-printable characters to the string. Any character with ASCII encoded value less than 32 is appended to the string
	StrOverflow	Add characters to overflow max size
Uint	UintEmpty	Replace by empty value
	UintAbsoluteMinusOne	Replace by -1
	UintAbsoluteOne	Replace by 1
	UintAbsoluteZero	Replace by 0
	UintAddOne	Add 1
	UintSubtractOne	Subtract 1
	UintMaxRange	Replace by maximum value valid for the parameter
	UintMinRange	Replace by minimum value valid for the parameter
	UintMaxRangePlusOne	Replace by maximum value valid for the parameter plus one
Opaque	OpaqueEmpty	Replace by empty binary string
	OpaquePredefined	Replace by predefined binary string with random length in the range of the valid minimum and valid maximum, containing a mixture of bytes with random value between 0 and including 255
	OpaqueOverflow	Add characters to overflow max size
Empty	EmptyPredefined	Replace by predefined value
	EmptyAbsoluteMinusOne	Replace by -1
	EmptyAbsoluteOne	Replace by 1
	EmptyAbsoluteZero	Replace by 0
Payload	PayloadEmpty	Replace with empty binary string
	PayloadPredefined	Replace with predefined binary string. Uses same rules as for OpaquePredefined
	PayloadAddNonPrintable	Add a non printable character. Uses same rules as StrAddNonPrintable

Table 4.1: Type-based Mutation Rules

Target	Tickets
Version	1
Type	3
Token Length	2
Code Class	3
Code Detail	3
Message Id	5
Token	5
Options	30
Payload	15

Table 4.2: Mutation target ticket distribution

4.7 Tested CoAP Attack Vector

The attack vector that is generally available to an attacker in CoAP servers is the message that communicated to the server. There is number of features that this experiment does not cover, that could possibly contain vulnerabilities. A reason for not testing these features is that it takes time to modify the fuzzer to cover them, and that the basics of the protocol is the most import piece of code to test.

Some notable features mentioned in the CoAP RFC [23] and elsewhere that this experiment does not explicitly test (that should be tested in production situations if available):

- Observe option
- HTTP-CoAP and CoAP-HTTP proxies
- DTLS
- CoAP over TCP, TLS, and Websockets (defined in RFC8323 [3])

4.8 Implementation

This section describes details about the practical implementation of the experimental suite.

Developing the experiment suite was done using C++ as the programming language with the only external tool being DynamoRIO for calculating code coverage. The creating of CoAP messages, mutations, communication was done using only C++ standard libraries.

DynamoRIO was invoked using a system command, thus creating a new process that uses the same environment as the experiment suite. This has both advantages as well as disadvantages compared to invoking DynamoRIO functionality directly from the fuzzer process. The main benefit is that it is easy to implement and configure as long as the correct path to the server binary is used. The disadvantages of this approach is a decrease in performance, which leads to fewer test inputs sent in a given time frame. After DynamoRIO is started as a new process, it runs and monitors the CoAP server as another process and writes the code coverage information to a file only once the CoAP server is terminated. The fuzzer sequentially sends the inputs over the local network and once it is finished it has to find the process of the CoAP server and terminate it. A problem with this approach is that there needs to be delays in the fuzzer to accommodate for the writing of code coverage logs and the general delay when communicating over the operating system. These inefficiencies are addressed in section 6.5.2.

The source code for this project and experiment can be found at `https://github.com/fliljeda/CoAP-Evo-Fuzzing-Msc`. An important detail about the source code is that it is written purely as a proof of concept for this report and there was no intention of developing it as a tool to use in production environments. As a result it will most likely not run properly on any other machine unless manual configurations and installments has been made. The source code is linked in this report for transparency reasons; any detail not mentioned in this section is observable in the git repository.

4.9 Running the experiment

Due to the manual work required to tailor the system to handle different servers, only a few were chosen to be tested. Looking at the work by Melo [17], there is a huge difference in robustness between applications due to the fact that there were 17 found bugs in one of the applications and few to none in others. To get some diversity to be able to draw comparisons, three server applications were chosen. One that had lots of bugs, one that a had a few bugs, and one that had few to none bugs. A requirement was also that it should be compatible with DynamoRIO. The chosen CoAP servers were:

- Canopus, written in Golang. Suspected of containing a large amount of errors. Found at `https://github.com/zubairhamed/canopus`.

Version e374f5b.

- Libcoap, written in C. Believed to have comparably few bugs. Found at `https://github.com/obgm/libcoap`. Version 102836c.
- Libnyoci, written in C. Bugs believed to exist but in modest numbers. Found at `https://github.com/darconeous/libnyoci`. Version 64cac86 tagged full/latest-release.

A server test can begin once initial configurations has been done for the tested server. This includes creating a customized seed file that contains example CoAP messages that can be parsed and sent to the server and changing the configuration file so the testing suite can identify the server binary to execute, the module to include in filtering of fitness logs (a method of increasing the accuracy of fitness calculations described in Section 4.3), and where to direct the logged output. Once the initialization is complete, a few practical test executions of the real tests are done to see if there are any crashes or failures. The failures monitored are the responses of the server and the fitness calculations. Responses from the tested server are logged as output and the server is expected to respond occasionally making it visible in the logged output. Fitness is monitored by observing the log file in real time to make sure it does not approach zero, as it would mean the server is not running correctly. If the initial testing looks promising, the real testing for the research is run by executing the main program of the experimental suite. The tests are then run as described in 4.2 autonomously using hard coded parameters to limit the size of each test.

The experiment parameters, such as number of generations and sizes of pools, were inspired by the experiment in the work by DeMott, Enbody, and Punch [7]. Some parameters differ from their experiment due to the structures of the algorithm components. For this experiment each server was tested twice, where each test ran for 100 generations. Each generation contained five pools, each pool contained five sessions each, and every session contained ten CoAP messages. The tests took about 3 hours each to complete on a machine running Ubuntu 18.04.2 LTS 64bit, with Intel® Core^{TM} i5-4210U CPU @ 1.70GHz x 4.

Every generation a fitness values of both the best Pool and the best Session were logged. Because the algorithm tends to focus survival on the most fit test cases it is more valuable to record the best performing in each generation, rather than an average, as it more accurately project the success of the algorithm. If a crash occurred, the session that caused the crash was saved for false positive testing after the experiment is complete. In cases where the server provided any error message, it was separately logged for a simple root cause analysis of any encountered bug.

Chapter 5

Results

This chapter presents the results gathered from the experiments described in section 4. The results consist of code coverage graphs representing the fitness curve of the evolutionary algorithm on each server, a display of the found bugs and errors, and a lightweight root cause inspection of the encountered bugs.

The fitness results of the two experiments performed on each server is seen in the graph figures of each server section. The Y-axes corresponds to the number of executed basic blocks for the current generation of the experiments. Two lines track the progressions of the most fit pool and the most fit session in each generation.

5.1 Canopus

Figures 5.1 and 5.2 shows the fitness graphs of the two experiments performed on Canopus. Both experiments started at roughly the same code coverage, at around 12200 basic blocks. The general progression of the experiments is a slight increase by roughly 100 basic blocks. Pool code coverage fluctuated greatly in both experiments and in Figure 5.1 for experiment 1 it causes the pool code coverage to start with the same coverage as it ends with. But the pool code coverage tends to increase in the generations after a fall has occurred, visible from generation 5 to 70. From generation 70 to 100 it seems to decrease slightly. Figure 5.2 for experiment 2 shows the same fluctuating pool code coverage but the positive progression of the coverage is more visible. The coverage seems to recover from falls quite quickly and structurally increase from generation 1 to generation 100, finishing with a pool code coverage of about 200 basic blocks higher than it started with.

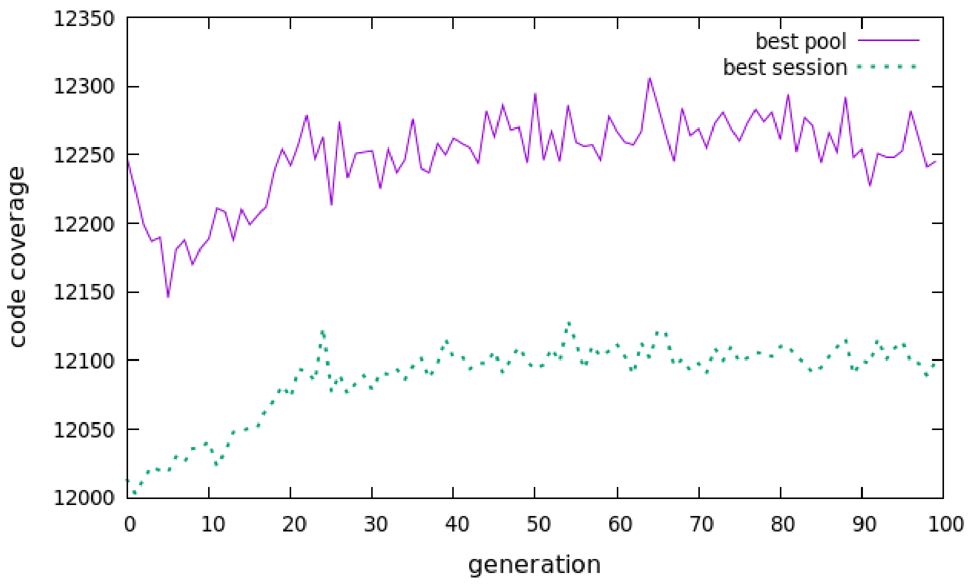

Figure 5.1: Canopus fitness graph experiment 1. Y-axis (code coverage) denotes the number of basic blocks covered in the server code. X-axis (generation) denotes the current iteration number of the evolutionary algorithm.

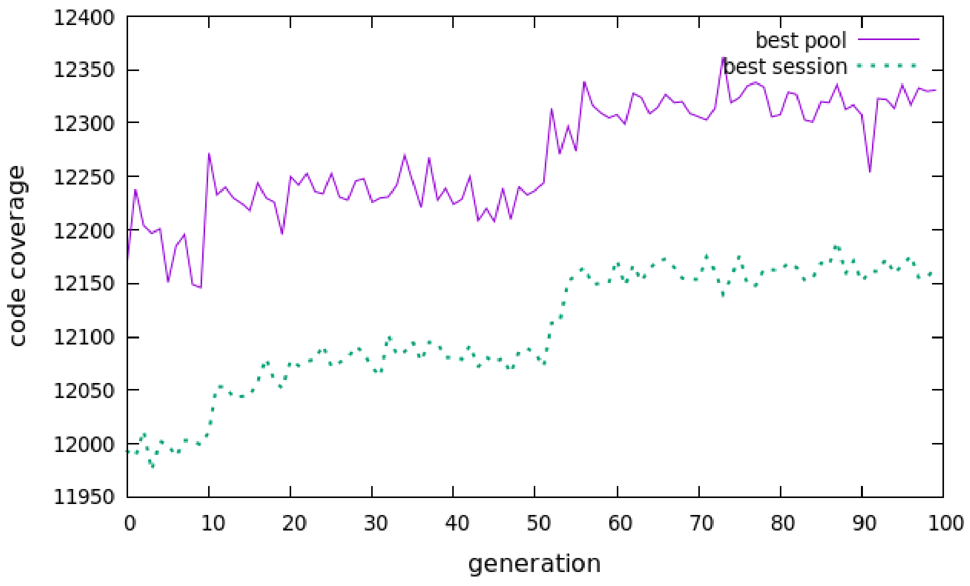

Figure 5.2: Canopus fitness graph experiment 2. Y-axis (code coverage) denotes the number of basic blocks covered in the server code. X-axis (generation) denotes the current iteration number of the evolutionary algorithm.

The curve of the session code coverage of both experiments have a trend of increasing as generations increase. Jumps of about 50 basic blocks are seen continually of both increases and decreases in code coverage. Certain jumps up are expected as combinations of mutations and CoAP messages triggers a new path of basic blocks. It is, however, unexpected to see decreases in code coverage for the best sessions. The reason for this being unexpected is that the best session of a pool skips the mutation process and is instead immediately copied to the new generation in a crossover operation. The most fit sessions would therefore be the same and cover the same code. This suggests that either the state of the server has changed or that the code coverage measurements' lack accuracy, either because of the tool DynamoRIO or the implemented parsing of logs.

5.1.1 Crashes and Errors

As expected of this CoAP server there were a large number of crashes that would hinder the server's availability causes by sending certain packets. Due to the unaddressed issues brought up by Melo [17] and current lack of response from the creator/s of the application this section will refrain from revealing certain details about the vulnerabilities and the packets that causes such issues, in an attempt to protect any usage of this specific CoAP server.

There were two different types of errors detected during the course of the two experiments. The golang server outputted one of the following two error identifiers before becoming unresponsive in an apparent crash:

- panic: runtime error: invalid memory address or nil pointer dereference
- panic: runtime error: slice bounds out of range

It is fairly simple to understand how the code breaks by analyzing the error messages; all *invalid memory address or nil point dereference* error messages originate from a single line of code which seems to happen because an unchecked dereferenciation of a null value. An interesting note here is that the error occurs in code that is meant to handle errors, and is also caused by erroneous error handling logic. The *panic: runtime error: slice bounds out of range* errors happens if the execution tries to read or write beyond the bounds of an array structure and in these experiments it originates from numerous locations in the source code. This happens due to failure to check the bounds of arrays/slices before writing to or reading from them.

A simple root cause analysis was performed on the malicious sessions. One error was caused by mismatching a declaration of a value and the actual value in the CoAP message, causing the *slice bounds out of range* error. The *invalid memory address or nil pointer dereference* occurred when a particular value was placed at a particular field in the CoAP message.

Another error worth mention that did not show during the runtime of these two experiments but appeared during testing and other stages before the final experiments was:

- fatal error: concurrent map writes

This error occurs when writing information to an object that is locked by an another resource. The root cause for this error is not known to the author of this experiment.

5.2 Libnyoci

The code coverage information for Libnyoci is seen in Figure 5.3 and Figure 5.4 for its first and second experiment respectively. Both figures have different coverage progression but there are common traits seen in both of them. Most apparent are the incremental spikes in best pool code coverage that quickly fades back to the the previous value. Lasting effects on code coverage is achieved from spikes as seen in Figure 5.4 at generations 5 through 10 and 25 through 35. Many of the smaller increases in code coverage of experiment 1 seen in Figure 5.3 are results of similar spikes. Both experiments end up with a code coverage of a little over 1170 basic blocks. But it is also visible that the pool code coverage started with around 1200 basic blocks in their very first generations, though both quickly plummeted 50 to 100 basic blocks in the first 10 generations. After the initial drop the code coverage follows a long-term increase to about 1170 basic blocks before growing stale, except for the spikes that is seen continuously in both experiments.

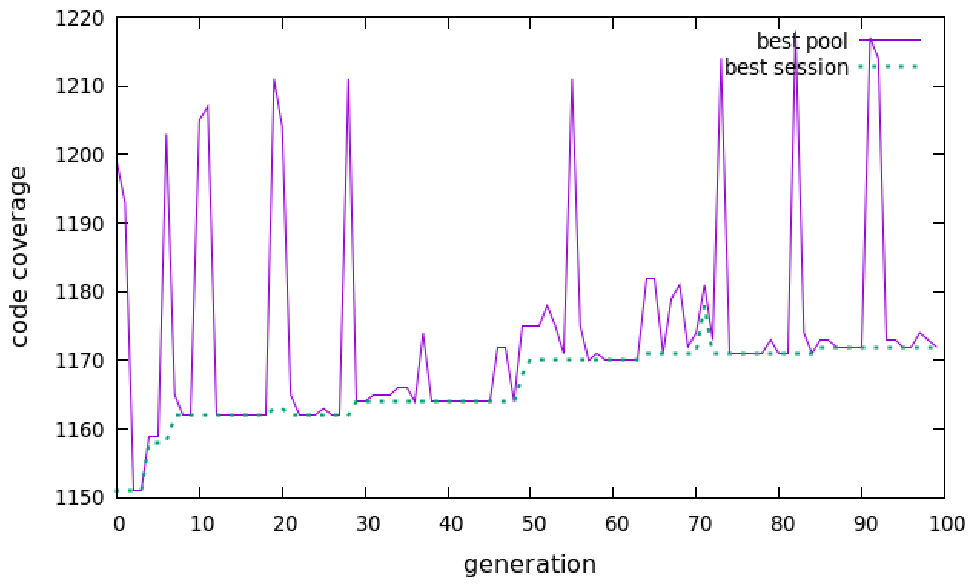

Figure 5.3: Libnyoci fitness graph experiment 1. Y-axis (code coverage) denotes the number of basic blocks covered in the server code. X-axis (generation) denotes the current iteration number of the evolutionary algorithm.

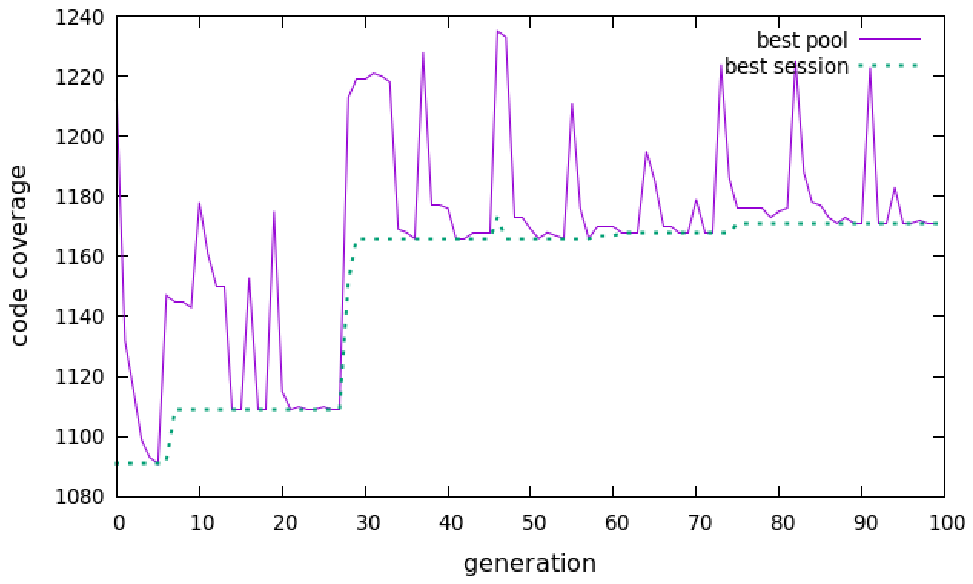

Figure 5.4: Libnyoci fitness graph experiment 2. Y-axis (code coverage) denotes the number of basic blocks covered in the server code. X-axis (generation) denotes the current iteration number of the evolutionary algorithm.

The session coverage employs a less dramatic progression than its pool counterpart. A few spikes can be seen in both figures and as is expected the coverage is maintained even after the spike increases, except for a few places such as generation 71 in Figure 5.3 and generation 46 in 5.4.

5.2.1 Crashes and Errors

Libnyoci crashed on several occasions throughout both experiments, causing the server to be unresponsive. The output from the server binary itself was minimal and contained no information about the crashes. As a result the logged packets from the crashes were tested on the server after attaching a debugging tool onto it. This revealed the location of the crashes and it was revealed that all the crashes experienced during the experiment originated from the same piece of code. To find out what caused the error, a packet was dissected bit by bit until no crash occurred and the liable section was found. It turned out that by only including a certain option number with a certain length, the server could crash by processing it. More details will not be provided as the bug still exist in the version labeled full/latest-release as of the moment of writing this. The bug has been patched in later versions using the information provided by Melo [17].

5.3 Libcoap

Figure 5.5 and Figure 5.6 shows the fitness graphs of experiments 1 and 2 on the CoAP server Libcoap, respectively. Both graphs shows a pool code coverage start at the lower half of 1600 and finish with a slight increase in code coverage, still getting a code coverage value in the 1600 range. There are spikes of increased code coverage in both that quickly plunges down to about the previously calculated code coverage. Although the spikes in Figure 5.5 look more dramatic than in Figure 5.6, the scale of the y-axes are different and the graphs shows that the spikes have similar increases in code coverage of about 50 basic blocks. The top of the spikes in experiment 1 have roughly the same value throughout the experiment but the session coverage increases in the first 50 generations and then stays at about 1620 basic blocks. Experiment 2 shows a more varied value for the pool code coverage spike increases and the highest value is 1750 basic blocks at generation 82, while the second highest is generation 73 covering 1723 basic blocks. The session coverage for experiment 2 increases greatly between generations 0 and 12, and then stays at about 1660 code coverage until a slight increase to 1674 at generation 64.

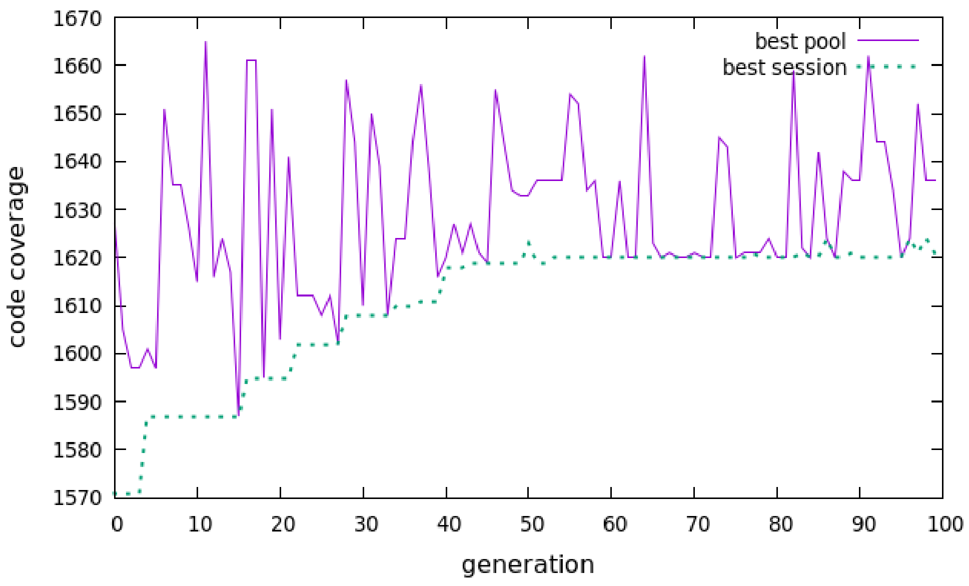

Figure 5.5: Libcoap fitness graph experiment 1. Y-axis (code coverage) denotes the number of basic blocks covered in the server code. X-axis (generation) denotes the current iteration number of the evolutionary algorithm.

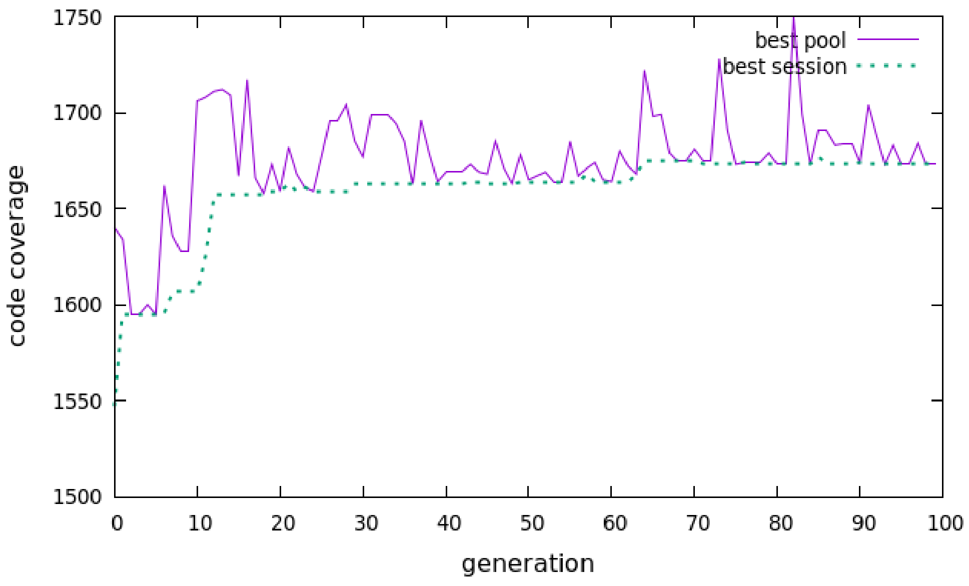

Figure 5.6: Libcoap fitness graph experiment 2. Y-axis (code coverage) denotes the number of basic blocks covered in the server code. X-axis (generation) denotes the current iteration number of the evolutionary algorithm.

5.3.1 Crashes and Errors

This server was expected to be resilient to fuzzy input and this seemed to hold true for both the experiments performed on it. None of the tested input managed to produce a crash that made the server unresponsive. However, by looking at the server error output generated by the binary an error message is present, stating that an assertion has failed and that the process was aborted. Assertions are mostly used for debugging purposes, and a failed one does not immediately point to vulnerabilities in the code considering the application works as intended following the error.

Chapter 6

Discussion

In this chapter a discussion is held regarding the connection between the research question, background information, methods used, and the results of the experiments to be used as a foundation for the conclusion.

To test whether or not evolutionary fuzzing is an improvement on simpler forms of fuzzing when testing CoAP server applications, an experimental suite containing a working evolutionary fuzzer was created from the ground up and then used on three different available CoAP server applications. By using existing software as targets, the results can be projected on real-life productions with greater confidence. A classification of improvement using a specific technique, is awarded if there exists arguments that would suggest that it is useful in replacement of the previously used technique. To create such arguments, this report uses measurements of code coverage, comparisons to previous fuzzing in terms of effectiveness and usability, and analyses of CoAP in conjunction with evolutionary fuzzing as important argumentative pillars.

6.1 Code Coverage

Code coverage can affect a fuzzy input's probability of encountering vulnerable code and is therefore the fitness value used in the evolutionary algorithms. As the evolutionary algorithm evolved each generation, it was intended that the fitness would increase to gradually cover more and more code. Because mutations are chosen randomly it is also expected that the fitness would increase in different magnitudes at different generations.

Looking at the figures of the two servers Libcoap and Libnyoci in Figures 5.3, 5.4, 5.5, and 5.6, a pattern is visible in the differences between the best pool coverage and the best session coverage. Pool coverage has spikes of high code coverage before often plunging down to the previously experienced code coverage. Session coverage increases occasionally but seems to stay at the increase code coverage. Since no modifications is done on the best session, this progression of the session code coverage is expected. The pool code coverage is measured based on the unique code covered by all sessions in it and there is always a possibility of a session to be removed or mutated between generations, impacting the unique code coverage of it; sometimes even negatively. While expected, it is not necessarily the progression the algorithm seeks to achieve. A good result would be for the best pool to remain on a high code coverage as it would mutate values that affect larger parts of the code. The results of the experiments seem to indicate that the evolutionary algorithm used is not suitable at maintaining an increasing pool coverage. A spike in code coverage is often quickly followed by a drop in the next generation or the one after that, suggesting that the session crossover operation performed on every generation may be the culprit causing the drops in code coverage. Session crossover prioritizes individually fit sessions and sessions with smaller (but potentially different) code path could be excluded from the new sessions created by the crossover. This means that operations with intention of increasing bug finding capability could actually be harming the collective pool coverage and thus also the real bug finding capability. The other operations have a chance of decreasing coverage too, but as they are applied less often they are not as likely to be the cause of the decreasing fitness in the results graphs.

In servers Libcoap and Libnyoci, the lowest points of the best pool coverage seem to increase in over generations and suggest some kind of progress. But it happens due to the best pool containing the best session, thus measuring the same code coverage as that session. An equality in code coverage between session coverage and pool coverage indicates that the best session covers a code superset of all other sessions' covered code in the same pool. Such a state in the evolutionary algorithm contains only one useful session as the rest does not affect the pool fitness. But assuming the evolutionary algorithm is applied to a fuzzer it does not mean that the rest of the sessions are useless, because the values in them may be different from the most fit session and trigger a crash in a section of code also covered by the most fit session.

Canopus shows fitness in Figures 5.1 and 5.2 that differs from the results seen from Libnyoci and Libcoap. The progression of fitness in both pool and session code coverage jumps up and down in almost every generation. A problem with this is that session code coverage should never decrease in fitness due to no mutations performed on the best session. This phenomenon also occurs on a few occasions in Libcoap and Libnyoci, but not with the continuity displayed by Canopus. The major difference between Canopus and the other two servers is that it is written in Golang instead of C. There could be problems in the accuracy of the code coverage tool DynamoRIO when analyzing Golang binaries, or there could be an issue in the code coverage log parsing. The cause for this behaviour is unknown at the time of writing this. The code coverage of pools and sessions always differs by a noticeable amount which is the sought after behaviour of using sessions and pools. However, skepticism should be maintained as this observation is only seen in one of the three servers; especially after suspecting the code coverage metrics to be moderately inaccurate. It is possible for the unique code coverage measurements for a pool to be distorted by inaccurate measurements of individual sessions. If the code coverage calculation mistakenly thinks the sessions covered different parts of the code it would measure the pool code coverage as a sum of the inaccurate measurements. Even with inaccuracies, the progression of average fitness increases with generations suggesting that the evolutionary algorithm manages to gain traction in code coverage. Figure 5.2 shows a better net change progression in pool code coverage than seen in 5.1 but session code coverage has a long term improvement in both.

A summary of the code coverage discussion so far is that the pool fitness progression over generations is not as effective and stable as hoped, and it may be due to the session crossover operation performed between generations. The Golang server Canopus may suffer from inaccurate code coverage measurements, but seems to have a similar net progression compared to the other servers. Due to the massive amounts of code coverage spike increases that quickly plummets to a lower value observed, especially in Libcoap and Libnyoci, it seems as if the fuzzer mutates inputs in an effective manner to cause the spike increases but some changes to the contents of the pools causes them to lose the progression. This suggests that different operations may increase the effectiveness of the evolutionary algorithm's progression when applied on CoAP servers.

6.2 Crashes

The purpose of any fuzzy testing is to encounter bugs during the testing. The quality and quantity of bugs found by a fuzzer is a good metric of its success and for this reason the encountered crashes will be discussed and compared to previous work on CoAP fuzzing

In the server Libcoap, the evolutionary fuzzer never detected any critical error during the two 100 generation experiments, and none in the test runs made before. Libcoap was expected to be resilient as it is a popular implementation of CoAP, with other servers recommend using the Libcoap client to test other server implementations. The results of this experiment neither disqualifies evolutionary fuzzing or proves absolute resilience for Libcoap. A specific path in the Libcoap code could have been untouched by the fuzzer, or just not covered with the correct vulnerable input. Maybe a longer running time of the fuzzer would have covered these cases. But as of this moment there is nothing indicating that the fuzzer would be able to find bugs in Libcoap.

Libnyoci was vulnerable to one specific input found by the fuzzer. The vulnerability caused the server process to crash and become unresponsive. Root cause analysis of the vulnerability showed no signs of attacks other than a denial of the service. This vulnerability was caused by a simple mismatch between a declared value and its actual value, making it an easy find by the fuzzer. It was triggered a multitude of times over the course of the 100 generations, in both experiments. The bug was fixed in the original code base due to being discovered by normal black-box fuzzing before this experiment, making it a non-unique find for the evolutionary fuzzer. There were also no indication that the bug was found more frequently with the growth of generations, as the crashes happened at random intervals during the experiments. Thus the crash discovered on Libnyoci does not indicate any improvements in bug finding capability when using evolutionary algorithms with fuzzing on CoAP.

Canopus is a far more broken CoAP server as several bugs were found during the course of the experiments. It crashed many times during the experiments, with three different error messages and two easily reproducible. The root cause analysis of the two reproducible crashes pointed towards failed error handling and the absence of correctness checks. It appears as if they were simple enough for any black-box fuzzing technique to find them as they could be triggered by changing a single part of a message without any relation to other values in the message. The third error triggered during the testing phases prior to the experiment may have been a more complex error since it involved writes on a locked object indicating a failure state in the server or failure due to a combination of values in the message. Unfortunately the packet logs were not available at the time, so the cause for the failure is unknown at this time.

None of the three servers produced any errors that would suggest a superior bug finding capability by using evolutionary algorithms in conjunction with fuzzing.

6.2.1 Previous CoAP Fuzzing

In the black-box testing by Melo [17] the fuzzy test inputs were generated by either generating random data, according to the protocol's specification, or mutating existing messages. The evolutionary fuzzer created in this research is both generating according to specification and by mutating existing inputs, as parts of the evolutionary algorithm. Much inspiration has also been sourced from Melo [17] such as the rules for generating parts of the CoAP messages. It is therefore intuitive to believe that the failure detection of both fuzzers would be of similar quality, but the results show that Melo [17] discovered 7 unique failures for Libnyoci, 15 unique failures for Canopus, and 1 failure for Libcoap. Comparing with the results of this research's fuzzer that found only 1 failure for Libnyoci, 3 for Canopus, and 0 for Libcoap. The versions of Canopus and Libnyoci were similar in both experiments and bugs were not fixed in the time between them. Such a big difference is alarming for this research as it could indicate a lack of quality in the implementation. One key difference between the two fuzzers is that the one used by Melo [17] generates a complete set of test cases in its generational engine and the one used in this research uses rules randomly. There is a possibility that certain bug-triggering values were never selected due to randomness and that a longer experiment with the evolutionary fuzzer would provide similar results. Another possible reason is that this research tested a comparably small portion of the functionality available in CoAP, as described in Section 4.7. The intuition of code coverage's relation to bugs could be applied; more obscure functions are less tested and therefore more prone to erroneous code.

Summarizing the comparisons between this research and the previous research it is clear that the work by Melo [17] has been more successful in bug discovery.

6.3 Constrained Application Protocol

CoAP was designed to be a lightweight and simple protocol to prevent errors caused by complex parsing. It is evident that fuzzing is still a useful method on such designs as errors were detected. Most errors existed in the options section of a CoAP message, although errors caused by values in the CoAP header were detected. As more functionality is being added to the protocol such as features described in Section 4.7 one can expect its complexity to increase. The experiments performed in this research did not cover enough of the available attack vector to fully test such applications. Fuzzing these additional functionalities could be a crucial step towards securing CoAP applications.

6.3.1 CoAP and Evolutionary Fuzzing

In the pioneering work by DeMott, Enbody, and Punch [7], a primary reason for using evolutionary fuzzing was to provide a method of learning the specification of the protocol and then using that knowledge to generate more effective test cases. In this research it was only used as a way to accomplish a more effective bug finding capability. It is therefore possible that the grey-box algorithm proposed by DeMott, Enbody, and Punch [7] is not optimal for the specific purpose of fuzzing small protocols such as CoAP. Perhaps a more white-box approach would have been more effective since it would have been more connected to the actual source code. There could also be benefits in changing the operations performed on each generation so that they affect the evolutionary algorithm in a more effective manner.

A benefit of using a evolutionary fuzzer is that it utilizes combinations of mutations to trigger certain vulnerabilities. For example there may be code snippets that are reached with the help of one variable but the bug is triggered by another variable's value. There could also be states in the application that are more vulnerable than other. It is important to understand if CoAP servers are prone to these kinds of vulnerabilities or not. CoAP messages have a structure that separates most fields from affecting each other during parsing. Due to the header being a set number of bits and the contents have an assigned location it is hard to imagine much difficulty in parsing it. Most values in the header only affects one part of the message and is unlikely to trigger bugs combined with other values. The message ID and token are pure data which is likely only stored in values and used for comparisons. Payload is mostly application specific and is hard to fuzzy test and evaluate properly. The option fields are the most likely to cause problems with combinations as they require a relatively complex parsing. But options are themselves mostly isolated from other options and are unlikely to affect each other in any significant way. This could prove to be a problem for the evolutionary fuzzer as its strengths are vastly reduced due to the structure of the protocol. None of the crashes encountered in the experiments performed in this researched were caused because of a combination of values, except for combinations of internal values of individual options.

A reason for having multiple messages in a session is to create a state in the server that would be more prone to crashes. But a general application layer protocol such as CoAP is unlikely to have any notable states affected by network packets. No error was traced to such behaviour in the experiments as all crashes could be triggered using only a single network packet.

6.4 Delimitations

A more detailed and complete root cause analysis was not performed as it is not directly relevant to the core of the research question. It is interesting and important to analyze what bugs appear in the code and draw parallels between applications to find a pattern. An error in the code could occur in similar forms in other parts of the code, posing an opportunity to prevent further crashes by patching them. Some errors could expose more serious vulnerabilities that would for example let an attacker take control of the device. But such analyses do not influence the answer to the research question directly and was therefore left out of the research.

Section 6.5 shines light on certain topics that were not dealt with properly in this research and should be considered in future research.

6.5 Future Research

This section explains the improvements that can be used to expand this research further.

6.5.1 Evolutionary Fuzzing On CoAP

There were a few important aspects in the study of evolutionary fuzzing that were left out because of time constraints. Because of the relatively underwhelming results of this research it could be useful to further investigate such features before consensus is reached regarding evolutionary fuzzing's effectiveness on CoAP applications.

A decision was made in this research to choose a suitable evolutionary algorithm and stick with it. There are both good and bad aspects of this decision. The benefit is that the research becomes more focused and time is spent on the core subject of testing applications rather than evaluating different algorithms for longer periods of time. The bad aspects of choosing just one algorithm before any experimental evaluation has been done is that there may exist better evolutionary algorithms that are more suited for this type of fuzzing. An expansion to this research would be to analyze the benefits and drawbacks of particular parts of an evolutionary algorithm when applied to a CoAP application.

The operations of the evolutionary fuzzer could benefit from further analyses as it is suspected that certain operations affects the evolutionary algorithm negatively in its quest to achieve higher code coverage. This also applies to the mutations performed on single CoAP messages. Many percentages and seed values were improvised using intuition in preparations for the experiments and some of them are most likely not optimal.

Allowing longer experiments that exceeds 100 generations may be important for the development of the evolutionary algorithms as many operations depend on randomness and may take many generations before certain positive changes appear. This research limited the generation cap to 100 only due to the time it took to run the experiments. This is addressed in detail in section 6.5.2. There should also be benefits to running more than two parallel experiments as the graphs for each server shown in Chapter 4 displays wildly different patterns. A greater variety and redundancy would make the results less biased towards potential anomalies.

6.5.2 Implementation Improvements

The delays in the experiments that are caused by starting the server, sending packets over the network interface, closing the process, waiting for DynamoRIO to write logs, and only then be able to parse the logs, should not be overlooked. All of the implemented delays combined results currently in a session of ten CoAP messages taking roughly five seconds to finish. And by using five pools each containing five sessions means that there are 25 sessions tested in each generation. 25 sessions with each taking five seconds results in a generation taking two minutes to complete. Every hour we manage to complete 30 generations using the current version of the implementation. This is arguably too slow for a testing method such as fuzzy testing that relies on multitudes of inputs to yield good results. The original version of the evolutionary algorithm used in fuzzy testing by DeMott, Enbody, and Punch [7] took roughly the same time to complete and they ran the algorithm for 100 generations with good results. It should therefore not be a major issue for the intended progress of the evolutionary algorithm, but the low number of tested inputs remains a problem for the completeness of the fuzzy testing.

Fixing the delays in the experiment suite should be possible given enough time and knowledge of the code coverage tool. The delays are caused currently by running DynamoRIO as a new process every time a server is started, because this is the most simple use of the tool. If the fuzzer was connected so that it communicates directly with the code coverage tool, information about the runtime code coverage could be extracted directly instead of writing to the file system. Another improvement could be to calculate code coverage without closing the server down between each session, eliminating most start and termination delays. A hidden performance benefit of this is that the process would be connected to fuzzer and there would be no need to scrape the process table to find the correct process of the server, eliminating an additional delay. If the servers source code is available it should be possible to connect directly to the servers packet handling functions and thus avoid using the network interface resulting in further speedup of the fuzzer.

Using the suggestions made in the previous paragraph, there should be little notable delays in the fuzzer and the process could reasonably be sped up to properly match the performance of an ideal fuzzer. These changes were not made in this project mostly due to time constraints.

6.5.3 Other Protocols

CoAP was selected as the targeted protocol as it has a potential for future widespread usage. There are other protocols that can possibly show greater potential for evolutionary fuzzing such as MQTT and XMPP. Their characteristics and behaviours may be vastly different to that of CoAP but this research can be used for inspiration and help in decision making when performing similar research.

6.6 Sustainability and Ethics

While the studied technique's intention is to break programs and could be used for malicious practices, it is still a study that would benefit the society as a whole. Revealing bugs in programs is equally important for the developers of the applications as for any malicious actors. IoT devices are often deployed in great numbers and a disruption of their services could prove catastrophic in the future. Testing techniques such as the one tested in this research are valuable additions to a developer's toolkit in that they can find the bug before anyone else.

Evolutionary fuzzing does not affect any other aspect than the robustness of certain applications.

Chapter 7

Conclusions

A fuzzer's primary goal is to expose bugs not found with manually created input and it was apparent that it worked with success in the master's thesis by Melo [17]. Applying the evolutionary fuzzing algorithm found in the paper by DeMott, Enbody, and Punch [7] to work on CoAP servers and running it for 100 generations showed no increase in encountered bugs. The amount of errors discovered in this research was lower than what was reported by Melo [17], though the difference may not depend on factors relevant to the evolutionary algorithm.

The intuition that higher code coverage of tests would yield a bigger chance of uncovering bugs in code is an important assumption in evaluating an evolutionary fuzzer. Code coverage progression was at the center of the experiments performed and the better the evolutionary algorithm became in terms of code coverage the better the chances would be to find bugs. The code coverage progression measured in the experiments turned out to be underwhelming in that high code coverage often lasted only a short while and the difference in values between the start and end of an experiment showed little progress.

Considering both the small number of bugs discovered with this method and the failed progression of the evolutionary algorithm it is clear that this research did not show any signs of improved fuzzy testing.

Further research on evolutionary algorithms, evolutionary operations, improving performance of the development suite, and the parsing of CoAP messages in servers can be done to more confidently reject the possibility of using evolutionary algorithms in CoAP fuzzing effectively.

Bibliography

[1] Kevin Ashton et al. “That ‘internet of things’ thing”. In: *RFID journal* 22.7 (2009), pp. 97–114.

[2] André Baresel et al. “The interplay between model coverage and code coverage”. In: *Proc. EuroCAST*. 2003, pp. 1–14.

[3] C. Bormann et al. *CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets*. RFC 8323. RFC Editor, Feb. 2018.

[4] Sergey Bratus, Axel Hansen, and Anna Shubina. “LZfuzz: a fast compression-based fuzzer for poorly documented protocols”. In: *Darmouth College, Hanover, NH, Tech. Rep. TR-2008* 634 (2008).

[5] Martin Büchi and Wolfgang Weck. *The Greybox Approach: When Black-box Specification Hide too much*. Citeseer, 1999.

[6] Sang Kil Cha, Maverick Woo, and David Brumley. “Program-adaptive mutational fuzzing”. In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 725–741.

[7] Jared DeMott, Richard Enbody, and William F Punch. “Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing”. In: *BlackHat and Defcon* (2007).

[8] Adam Doupé et al. “Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner”. In: *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX, 2012, pp. 523–538. ISBN: 978-931971-95-9. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/doupe.

[9] *DynamoRIO website*. https://www.dynamorio.org/. Accessed: 2019-04-16.

[10] Patrice Godefroid, Michael Y Levin, and David Molnar. “SAGE: whitebox fuzzing for security testing”. In: *Communications of the ACM* 55.3 (2012), pp. 40–44.

[11] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. “Automated Whitebox Fuzz Testing.” In: *NDSS*. Vol. 8. 2008, pp. 151–166.

[12] Rody Kersten, Kasper Luckow, and Corina S Păsăreanu. “POSTER: AFL-based Fuzzing for Java with Kelinci”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2017, pp. 2511–2513.

[13] Shancang Li, Li Da Xu, and Shanshan Zhao. “The internet of things: a survey”. In: *Information Systems Frontiers* 17.2 (Apr. 2015), pp. 243–259. ISSN: 1572-9419. DOI: 10.1007/s10796-014-9492-7. URL: https://doi.org/10.1007/s10796-014-9492-7.

[14] Y. K. Malaiya. “Antirandom testing: getting the most out of black-box testing”. In: *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE’95*. Oct. 1995, pp. 86–95. DOI: 10.1109/ISSRE.1995.497647.

[15] Richard McNally et al. *Fuzzing: the state of the art*. Tech. rep. DEFENCE SCIENCE and TECHNOLOGY ORGANISATION EDINBURGH (AUSTRALIA), 2012.

[16] Bruno da S Melo, Paulo Lıcio de Geus, and André A Grégio. “Robustness Testing of CoAP Server-side Implementations through Black-box Fuzzing Techniques”. In: ().

[17] Bruno da Silva Melo. “Robustness testing of CoAP implementations and applications through: Teste de robustez de implementações e aplicações utilizando CoAP por meio de técnicas de fuzzing”. MA thesis. Universidade Estadual de Campinas, Instituto de Computação, Campinas, SP, 2018.

[18] Barton Miller, Louis Fredriksen, and Bryan So. “An empirical study of the reliability of UNIX utilities”. eng. In: *Communications of the ACM* 33.12 (1990), pp. 32–44. ISSN: 1557-7317.

[19] Charlie Miller, Zachary NJ Peterson, et al. “Analysis of mutation and generation-based fuzzing”. In: *Independent Security Evaluators, Tech. Rep* (2007).

[20] P Oehlert. “Violating assumptions with fuzzing”. eng. In: *IEEE Security & Privacy* 3.2 (2005), pp. 58–62. ISSN: 1540-7993.

[21] Sanjay Rawat et al. "VUzzer: Application-aware Evolutionary Fuzzing." In: *NDSS*. Vol. 17. 2017, pp. 1–14.

[22] Z. Shelby. *Constrained RESTful Environments (CoRE) Link Format*. RFC 6690. `http://www.rfc-editor.org/rfc/rfc6690.txt`. RFC Editor, Aug. 2012. URL: `http://www.rfc-editor.org/rfc/rfc6690.txt`.

[23] Z. Shelby, K. Hartke, and C. Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. `http://www.rfc-editor.org/rfc/rfc7252.txt`. RFC Editor, June 2014. URL: `http://www.rfc-editor.org/rfc/rfc7252.txt`.

[24] Ari Takanen, Jared D. Demott, and Charles Miller. *Fuzzing for Software Security Testing and Quality Assurance*. eng. Artech House information security and privacy series. Norwood: Artech House, 2008. ISBN: 9781596932142.

[25] Lu Tan and Neng Wang. "Future internet: The internet of things". In: *2010 3rd international conference on advanced computer theory and engineering (ICACTE)*. Vol. 5. IEEE. 2010, pp. V5–376.

[26] P. Tonella and F. Ricca. "A 2-layer model for the white-box testing of Web applications". In: *Proceedings. Sixth IEEE International Workshop on Web Site Evolution*. Sept. 2004, pp. 11–19. DOI: `10.1109/WSE.2004.10012`.

[27] Benjamin Tyler and Neelam Soundarajan. "Black-Box Testing of Grey-Box Behavior". In: *Formal Approaches to Software Testing*. Ed. by Alexandre Petrenko and Andreas Ulrich. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 1–14. ISBN: 978-3-540-24617-6.

[28] Ilja Van Sprundel. "Fuzzing: Breaking software in an automated fashion". In: *Decmember 8th* (2005).

[29] Spandan Veggalam et al. "Ifuzzer: An evolutionary interpreter fuzzer using genetic programming". In: *European Symposium on Research in Computer Security*. Springer. 2016, pp. 581–601.

[30] M. Vieira, N. Laranjeiro, and H. Madeira. "Benchmarking the Robustness of Web Services". In: *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*. Dec. 2007, pp. 322–329. DOI: `10.1109/PRDC.2007.56`.

[31] Joachim Wegener et al. “Testing real-time systems using genetic algorithms”. In: *Software Quality Journal* 6.2 (June 1997), pp. 127–135. ISSN: 1573-1367. DOI: 10.1023/A:1018551716639. URL: https://doi.org/10.1023/A:1018551716639.

[32] Zhi-Kai Zhang et al. “IoT security: ongoing challenges and research opportunities”. In: *2014 IEEE 7th international conference on service-oriented computing and applications*. IEEE. 2014, pp. 230–234.

TRITA -EECS-EX-2019:703