Grammar-Aware MQTT Fuzzing

A New Fuzzing Strategy for the Security Testing of MQTT Broker Applications

Maksymilian Gotkowicz

Supervisor: Dr. Lucas Cordeiro

A Third Year Project Report for the degree of Bachelor of Science

Department of Computer Science
The University of Manchester
United Kingdom
May 2022

Grammar-Aware Fuzzing of the MQTT Protocol

Maksymilian Gotkowicz and Lucas Cordeiro

The University of Manchester

May 2022

Abstract

In recent years, the prominence of Internet of Things (IoT) networks consisting of vast amounts of low-power, low-memory internet-connected devices has given rise to a new attack surface for attackers to exploit. Vulnerabilities found in IoT communication systems can give rise to exploits, providing lateral movement into wider organisational IT infrastructures, with potentially devastating consequences. In this final year project, we focus on the testing of the MQTT Protocol - a lightweight communication protocol tailored for use in IoT devices. Significant contributions in this area have been made thanks to the development of the MultiFuzz packet-aware fuzzer, a fork of the AFL fuzzer that allows for testcase generation tailored to publish-subscribe protocols thanks to a fuzzing strategy that segregates inputs into packet entities. We propose a new custom mutator as an additional module to the more optimised AFL++ fuzzer, which implements both packet awareness and MQTT grammar according to the OASIS MQTT specification in order to generate more relevant testcases for input into MQTT Broker applications, thus resulting in faster path discovery compared to current state-of-the-art solutions and the potential of new vulnerability discovery programs that accept MQTT packet inputs.

Contents

1 Introduction **4**
- 1.1 Motivation . . . 4
- 1.2 The Problem . . . 4
- 1.3 Objectives . . . 4
- 1.4 Current Contributions . . . 5
- 1.5 Impact of COVID-19 . . . 5
- 1.6 Report Structure . . . 5

2 Background **6**
- 2.1 The Internet of Things . . . 6
- 2.2 The MQTT Protocol . . . 6
- 2.3 Software Testing . . . 7
  - 2.3.1 Black-Box Testing . . . 7
  - 2.3.2 White-Box Testing . . . 8
  - 2.3.3 Gray-Box Testing . . . 8
- 2.4 Fuzzing . . . 9
- 2.5 AFL and AFL++ . . . 10
  - 2.5.1 AFL: Fuzzing Process . . . 10
  - 2.5.2 AFL++ . . . 10
- 2.6 The EBF Tool . . . 11
- 2.7 Other Relevant Fuzzers . . . 12

3 Design and Implementation **13**
- 3.1 Project Methodology . . . 13
- 3.2 MQTT Packet Structure . . . 13
  - 3.2.1 Packet Types, Typical Communication and Grammar . . . 13
  - 3.2.2 MQTT Packet Format . . . 14
  - 3.2.3 Further Packet Grammar . . . 15
- 3.3 Fuzzing Environment . . . 16
  - 3.3.1 Desockmulti . . . 16
  - 3.3.2 AFL++ Environment . . . 18
- 3.4 Mutation Strategy . . . 19
  - 3.4.1 Create Entirely New Seed . . . 20
  - 3.4.2 Add A Packet Onto the Current Desockmulti Seed . . . 20
  - 3.4.3 Remove Random Packet from Current Seed . . . 20
  - 3.4.4 Change Desockmulti Accept/Connect Number . . . 21
- 3.5 Side Endeavor: Vulnerability Research in MQTT Clients . . . 21

4 Evaluation **22**
- 4.1 Evaluation of Mutation Strategies . . . 22
- 4.2 Testing on MQTT Broker Implementations . . . 22
  - 4.2.1 Testing Environment . . . 22
  - 4.2.2 Testing Results . . . 22
- 4.3 Evaluation of Results . . . 23
- 4.4 Threats to Validity . . . 24

5 Conclusion **24**
- 5.1 Project Summary . . . 24
- 5.2 Further Improvements . . . 25
  - 5.2.1 Most Significant Improvements . . . 25
  - 5.2.2 Other Potential Improvements . . . 25

Acknowledgements

This project has been conducted as part of the Systems and Software Security research group at the University of Manchester, with the support of Dr. Lucas Cordeiro, PhD Candidates Fatimah Aljafaari and Rafael Menezes and Research Associate Edoardo Manino, whom I wish to thank greatly for the continued support in the understanding of software testing and verification. The entirety of this project is built upon the infrastructre of the AFL++ fuzzer [13], developed by Mark Hause, Heiko Eißfeldt, Andrea Fioraldi and Dominik Maier - a fork of the original AFL tool [49] developed by Michał Zalewski. A huge thanks has to also be made to Yingpei Zeng, Mingmin Lin, Shanqing Guo, Yanzhao Shen, Tingting Cui, Ting Wu, and Qiuhua Zheng for the development of the open-source MultiFuzz [52] tool, which has served as an inspiration for this project, as well as the 'desockmulti' desocketing tool [53], which has been used extensively in this project in order to simulate multiparty MQTT connections.

1 Introduction

1.1 Motivation

As the cost of semiconductors has fallen throughout the last two decades, the large-scale use of low-power internet-connected devices has significantly increased, known collectively as the Internet of Things (IoT) [5]. These devices can range from thermostats to kettles to key card locks, and often collectively produce large amounts of data from sensors, actuators and other such devices. This produced data is later transmitted to a larger main network to be processed, usually with the help of a 'broker' intermediary. Especially in the case of IoT devices being used in security systems, it is necessary to use protocols with strong cryptographic capabilities and a reliable broker to ensure data confidentiality and integrity upon transmission, preventing the possibility of attacks through data interception and/or manipulation. Due to the limitations of IoT devices both in memory and processing power, the use of standard desktop protocols is infeasible, and thus, tailored lightweight communication protocols such as MQTT (Message Queue Telemetry Transport) [19] have been developed specifically for use in such devices. Due to the fairly recent widespread adaptation of IoT devices, many of these lightweight protocols have not been analysed as rigorously as those used in standard desktop/mobile devices, with new vulnerabilities constantly being discovered in many implementations [9]. According to the Eclipse Foundation [15], the MQTT protocol is now the most widely-adopted communication protocol in IoT devices, and thus it has become crucial to make sure that popular implementations of this protocol are extensively tested for vulnerabilities.

1.2 The Problem

Indeed, it has been found that there are limitations to current fuzzing techniques for the MQTT protocol due to the lack of structure-aware seed development [3].

Significant progress has been made in the form of fuzzers designed specifically for the testing of MQTT, however, the only solution that implements a state-of-the-art fuzzer, MultiFuzz [52], provides a more generic solution applicable to a variety of publish-subscribe protocols, and does not implement the MQTT's packet structure or grammar, leasing to a lot of irrelevant testcases being produced. In conjunction with the more optimised AFL++ fuzzer [13] and possibly more effective testcase generation thanks to the EBF tool's [1] model checker, we wish to devise a more effective fuzzing strategy than current state-of-the-art solutions, combining the packet awareness seen in MultiFuzz with MQTT grammar.

1.3 Objectives

The key objective set out to achieve in this project is to improve the EBF [1] tool's effectiveness in bug detection in the MQTT protocol, therefore, two main goals can be derived for the project:

- Improve the speed of path discovery in fuzzing applications that accept MQTT packet inputs compared to current state-of-the-art solutions
- Discover new vulnerabilities in various applications implementing the MQTT Protocol

1.4 Current Contributions

Many highly efficient generic fuzzers have currently been developed, with LibFuzz [36] and American Fuzzy Lop (AFL) [49] being the most popular solutions currently in use. AFL++ [13], a fork of AFL, has also recently been developed, and contains many improvements over AFL such as more effective instrumentation, and most notably, support for custom modules. AFL++'s support for custom modules, including custom mutators has made it the obvious choice for a platform to base the project on.

The most promising MQTT fuzzing effort has been made in MultiFuzz [52] - an altered fork of AFL tailored to fuzzing pub-sub protocols such as MQTT. MultiFuzz utilises a custom packet-aware fuzzing strategy alongside a tailored desocketing tool simulating multiparty connections and has proven to be significantly more effective at fuzzing MQTT applications than AFL alone, however, as previously mentioned, the tool does not implement MQTT grammar, which is one of the key aims of our project.

In order to further improve the quality of our results, we aim to run our custom mutator with AFL++ on the back of a more robust verification tool, EBF. Encryption-BMC and Fuzzing tool (EBF) [1] utilises additional program analysis in the form of the ESBMC tool [16] to generate more effective initial seeds for AFL++ and a custom LLVM pass alongside the TSAN tool [40] to better bugs specific to multithreaded implementations. Although our project focuses on MQTT brokers that primarily do not utilise multithreading, the additional analysis may prove useful in fuzzing future implementations that do take advantage of it. Other fuzzers focusing on fuzzing multithreaded programs such as MUZZ [8] and ConFuzz [32] have also been developed, but are unfortunately either closed-source (MUZZ) or not as robust due to the use of a "Dumb fuzzer" rather than AFL (ConFuzz).

A grammar-based MQTT fuzzer, MQTTGRAM [37] has also been developed, which provided more realistic fuzzing, even going as far as setting probabilities of different control packets, however, the tool does not utilise a state-of-the-art solution such as MultiFuzz with AFL, and the fuzzer remains closed-source, so cannot be analysed. Another popular contribution is F-Secure's mqtt_fuzz [12] - a basic open-source fuzzer consisting of a python script sending different types of MQTT control packets to a server. Other implementations include [7] and [43], all presenting custom closed-source implementations of MQTT fuzzers.

1.5 Impact of COVID-19

Although the impact of the COVID-19 pandemic may have not been as substantial as seen in the year prior due to various lockdown and restriction procedures, the pandemic has still produced a significant impact on the overall project. Most notably, catching a heavy case of the virus caused progress on the project to halt for almost a month, thus slowing down the pace of the research conducted.

1.6 Report Structure

In this report, we will first provide an overview of software testing and verification, and an introduction to fuzzers. We then provide a deeper insight into the AFL and AFL++ fuzzers and their architecture, as well as a brief look at other fuzzing implementations and the anatomy of the EBF tool. We then provide a detailed breakdown the MQTT Protocol, looking at its packet types, typical communication, packet construction and grammar. This is followed by a brief synopsis of the 'desockmulti' desocketing tool [53] used in the project, and then a wider presentation of our testing environment and architecture. Our custom mutation strategy is then presented in detail, followed by a small section covering other noteworthy research conducted throughout the project. Finally, we present an evaluation of our tool, results compared to competing tools in testing MQTT brokers, and a conclusion of the project with suggested improvements and next steps.

2 Background

2.1 The Internet of Things

In 1999, the term 'Internet of Things' was first coined by Kevin Ashton to describe the integration of RFID tagging systems in corporate supply chains to the internet as a means of automating tracking of goods.[4] The term, first used as a catchphrase to gain corporate attention, has since risen to prominence when describing different physical objects with sensors that collect data, and embedded devices that communicate this collected data over the internet. Smart traffic lights, smart central heating system sensors (e.g. Hive [18]) and wearables (e.g. FitBit [14]) are all examples of Internet of Things (IoT) devices commonly used today. Increased demand and supplier focus, monumental advances in efficient low-power processing (such as the the development of the ARM Cortex-M chips) [5] and decreased semiconductor costs in the last decade [31] have all been key factors in the recent rapid growth of IoT devices, with estimates of around 12.3 billion connected devices as of 2021.[42]

With mainstream adoption of any technology, comes a new attack surface that malicious actors will try and exploit, and so has been the case with IoT devices, with the number of attacks worldwide escalating to 1.51 billion in 2021 according to Kaspersky [10]. Especially in the case where aforementioned IoT devices form part of security or essential monitoring systems (such as door locking systems, IP cameras or health monitoring systems) or in cases where lateral movement from IoT Devices to broader IT networks is possible, such attacks can have devastating consequences. Notable examples include hackable cardiac devices from St Jude [22] and the Mirai IoT Botnet [48].

2.2 The MQTT Protocol

One of the current key challenges in IoT communication is protocol security [28] - a vulnerable protocol (or poor implementations of the protocol, such as exploitable client or broker applications) can serve as an easy entry point for an attacker to manipulate or gain control of an IoT network. According to the Eclipse Foundation December 2021 survey, the MQTT (Message Queue Telemetry Transport) protocol has become the most widely-utilised protocol for communication in IoT devices, with 44% of surveyed devices utilising it for sending and receiving messages [15], therefore, it is crucial that this protocol is thoroughly tested for any vulnerabilities that could potentially be exploited.

For communication using MQTT, at least two parties must be present - a client (or multiple clients) and a message broker (server). MQTT uses the publish-subscribe model, in which clients first connect to a server, and then have three key actions [29]:

- **Publish** - A client publishes a sends a piece of data to the broker with an associated topic name
- **Subscribe** - A client subscribes to a certain topic, all messages published with this topic name will be relayed back to the client by the broker.
- **Unsubscribe** - A client unsubscribes from a topic that the client had previously subscribed to.

In practical implementations, IoT devices serve as 'publishers' and send data with a given topic to the broker, whereas devices such as computers used for processing this data act as 'subscribers' - the MQTT broker relays data from the publishers to the subscribers. However, in theory, any client is capable of both publishing data and subscribing to topics.

Other packet types can be sent, and more detailed overview of the protocol grammar will be presented in section 3.2 of the report, however, these three actions construct the large majority of messages that the client sends to the broker. The data that is being sent alongside the publish messages and the topic naming structure is entirely dependent on the program with which the MQTT protocol is being used.

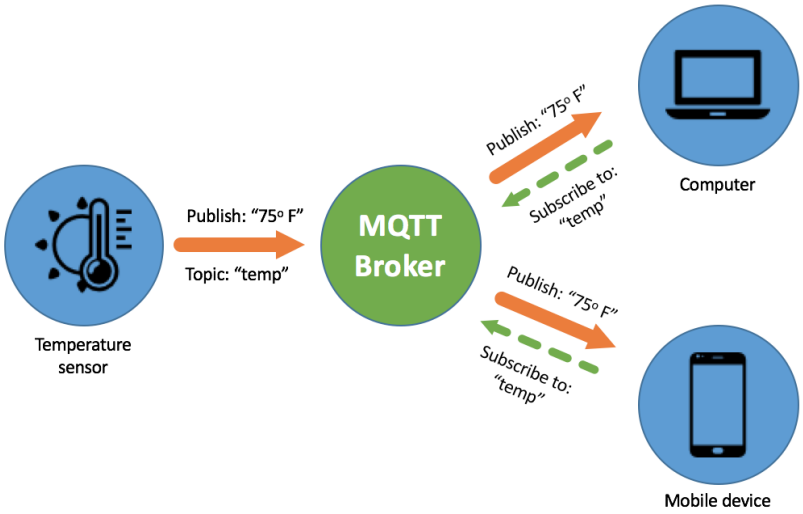

Figure 1: A basic overview of MQTT client-broker communication [30]

Rather than focusing on the programs that utilise MQTT, we look at implementations of the protocol itself - we will test the security of different MQTT Broker implementations such as Mosquitto [25] and NanoMQ [11] through the discovery of potentially harmful inputs that could cause the broker to crash, potentially exposing a vulnerability (e.g. a buffer overflow) that would allow for an attacker to manipulate/disrupt communication, discover otherwise inaccessible data, or even run arbitrary code. Some examples of such vulnerabilities have already been discovered in the past. [20][9]

Section 3.5 of the report will also cover some MQTT client testing done throughout the project and the results found via this process.

2.3 Software Testing

2.3.1 Black-Box Testing

Historically, there have been several methods of discovering vulnerabilities in software. By far the most popular type has been black box testing, in which inputs are fed into the Program Under Test (PUT) with no knowledge of the inner workings or structure of the PUT - this type of testing is often seen in software development pipelines in the form of various automated testing tools such as Selenium [38], which is used for testing web applications. This style of testing can be beneficial in catching common bugs and vulnerabilities, however are limited to the scenarios that the quality assurance team is able to conceptualise and script, and thus more advanced vulnerabilities can often get overlooked or may take too long for black-box testing software to catch in a reasonable time frame.

2.3.2 White-Box Testing

The other side of the coin is white-box testing, in which the testing software finds vulnerabilities by analysing the source code of the PUT. Unit tests and integration tests are the most basic examples of this, where developers write tests tailored to the code they have written to make sure their code behaves as intended, however, these tests are as good as the people who write them. More advanced white-box testing techniques allow for the discovery of vulnerabilities that developers may not have accounted for, these include [24]:

- **Static Analysis** - Analysis of source code without actually executing it to try and find errors before compilation. This usually involves techniques such as data flow analysis and lexical/grammar analysis that is nowadays often performed automatically by IDEs such as Visual Studio Code [26] and Eclipse [17] as well as many modern compilers such as GCC [34] and Clang [23]. Additional tools for creating more tailored static analysis such as CodeQL [33] and SonarQube [44] are now becoming increasingly popular in DevOps pipelines. Static analysis meets its limitations with more complex and modular code, where interaction between many different systems cannot be effectively analysed without running the code. Furthermore, more complex bugs such as data race in multithreaded programs cannot be detected through static analysis.

- **Dynamic Analysis** - Analysis of source code via additional instrumentation (monitoring) while executing it on a real or virtual CPU. Known examples include Intel Inspector and Clang ThreadSanitizer, the latter of which has been used in the project and will further be mentioned in section 3.5. These tools can be very effective in finding a variety of vulnerabilities from misused memory allocation to multithreading bugs, but often require a good set of test inputs and a specialised testing environment, making it hard to scale up for more complex programs.

- **Symbolic Execution** - Tools that analyse the inputs of a source code and 'symbolise' them, converting them into a set of satisfiability problems [21] to be solved by constraint solvers. This technique is by far the most precise, going further than software testing into the realm of software verification, in which programs are proven to be correct/safe. However, with more complex programs with many inputs, the amount of paths grows exponentially, meaning that poor scalability is a major drawback of this technique.

- **Model Checking** - Tools that create a finite computational model of the source code and check that a desired logical property holds in this model by exhaustive state space search. [6] Similar to Symbolic Execution, this is a verification technique rather than a testing technique and is thus a lot more accurate, however, suffers from the same state-space explosion problem as symbolic execution in large and complex systems, again making scalability an issue.

2.3.3 Gray-Box Testing

In between black-box and white-box testing, we have gray-box testing, in which some lightweight program analysis is performed, but without any heavy processing such as constraint analysis or other forms of verification.[51] A good example of this is the AFL (American Fuzzy Lop) fuzzer [49], which will be covered further in the next section.

2.4 Fuzzing

First proposed by Barton Miller in 1988, fuzzing is currently by far the most popular automated vulnerability discovery technique [45]. It involves the large scale execution of automatically generated testcases into the PUT in order to force a crash or abnormal program state. State-of-the-art solutions such as AFL often run multiple instances of the program simultaneously in different fork processes for faster results.

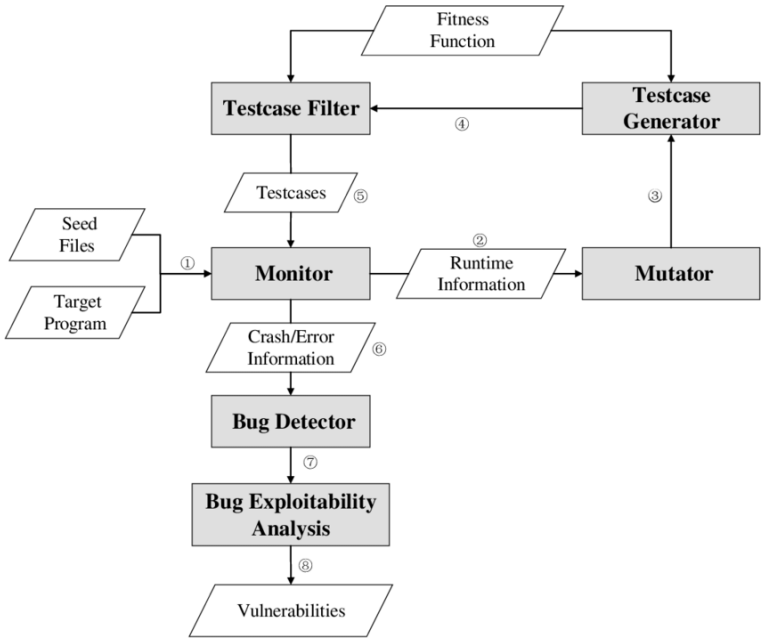

Figure 2: The process flow of a mutation-based fuzzer. [46]

Figure 2 portrays the typical structure of a mutation-based fuzzer. We start with a PUT and a set of valid 'seed' inputs to begin with. The fuzzer starts off by executing the PUT with the seed inputs, and then uses a mutator function to alter the inputs , generating new testcases in an effort to force different program behaviour. The fuzzer uses a variety of monitoring techniques to look at the program behaviour, and uses the results from this monitoring to prioritise testcases that change program behaviour and run them through an new cycle of mutations to generate new testcases. Whenever execution of the program using one of the testcases as an input causes a program to crash, the information is printed and can then be sent to a bug detector to analyse a potential vulnerability in the program. This process continues indefinitely, and in practice, the fuzzing process can take hours, and even days, often on machines with large amounts of computational power.

Fuzzers can fit all three testing categories from white-box to black-box, with some doing no program analysis or monitoring and simply running many slightly altered inputs (so-called 'dumb fuzzers' [27] such as F-Secure's mqtt_fuzz [12]), and others (such as FuseBMC[2]) performing deep symbolic execution to generate higher quality inputs. Fuzzers that perform at least some program analysis (i.e. aren't black-box) are known as 'smart/intelligent fuzzers' and can typically be categorised into two types [24]:

- **Mutation-Based** - Only a set of initial valid inputs is required - new testcases are generated by mutating (slightly altering) existing inputs
- **Generation-Based** - Testcases are generated based on a given configuration file or according to a specific file format, as well as possibly through source code analysis.

As further analysed in [24], due there being no requirement of manual program analysis to start fuzzing, mutation-based fuzzers are generally more applicable. Current state-of-the-art solutions are mostly mutation-based due to their wide adaptability to a variety of different PUTs without any additional configuration. Furthermore, fuzzers can be categorised into coverage-guided and directed.

- **Coverage-Guided** - Fuzzers that prioritise testcases that cover new functions and paths that haven't previously been executed.
- **Directed** - Fuzzers that prioritise testcases that cover specified areas of code that a user wants to be tested rigorously.

By far the most popular types of fuzzers are coverage-guided gray-box fuzzers. A prime example is the AFL (American Fuzzy Lop) fuzzer [49] which only uses lightweight instrumentation in the form of injecting additional code at each conditional 'jump' instruction (e.g in an `if` statement) to track which branches of the PUT have been executed. Throughout the fuzzing process, AFL keeps track of code coverage by keeping a store of which branches have been executed - no further code analysis is done beyond this. AFL is currently the most extensively used fuzzer, and this project will focus on better adapting an improved fork of AFL (AFL++) to generate higher quality testcases when testing programs that accept MQTT packets (such as MQTT Brokers).

2.5 AFL and AFL++

As mentioned in the previous section, the focus of the project is improving a fork of the AFL fuzzer for better MQTT testcase generation. In turn, the inner workings of the AFL fuzzer need to be introduced to portray where said improvements will be made.

2.5.1 AFL: Fuzzing Process

Before a program can be fuzzed by AFL, it must first be compiled with AFL's custom instrumentation in order to inject monitoring code at `JUMP` statements. AFL has a built in module ('afl-cc') which uses Clang [35] or GCC [34] to compile the source code alongside its own instrumentation. Once compiled with the instrumentation, the 'afl-fuzz' module creates a set of mutated inputs based on the next item in the input queue (which will start off with an initial set of inputs given by the user). Each mutated input will then be executed on the PUT in a separate forkserver process and during execution, the additional instrumentation will alter a 'coverage bitmap' in the shared memory each time a branch is executed. The coverage data is then fed into a result analyser, which sends any mutated data that causes new branch hits into the input queue, marks it as 'interesting', and writes both the testcase and the fuzzing results into a file in memory. This process repeats indefinitely.

2.5.2 AFL++

Rather than standard AFL, the project will be based upon AFL++ - a fork of AFL with various optimisations and improvements. Improvements include support for testcase minimisation, better LLVM compiler support, and most notably, easy support and helper modules for custom mutators, which is the main artefact of the project. The AFL fuzzing process structure has been retained by AFL++, and the use of a custom mutator such as the one in this project involves a simply the export of an environment variable called 'AFL_CUSTOM_MUTATOR_LIBRARY' with the path to the custom mutator. Customisable helper modules such as 'custom_mutator_helpers.h' also simplify the process of effective custom mutator development.

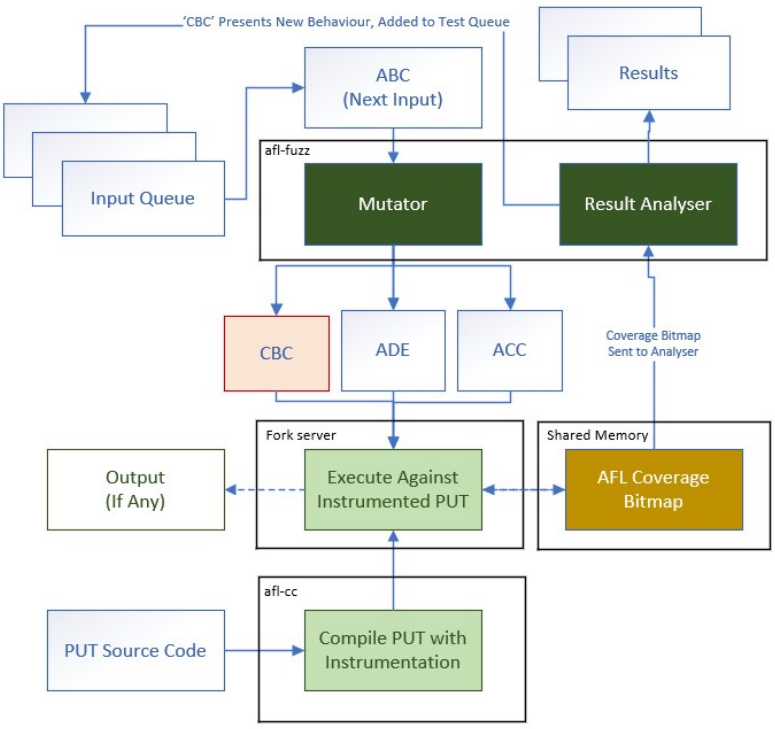

Figure 3: The structure of the AFL Fuzzer.

2.6 The EBF Tool

The Encryption-BMC and Fuzzing tool created by Fatimah Aljafaari, Rafael Menezes, Dr. Lucas Cordeiro and Dr. Mustafa Mustafa [1] is a tool that builds upon AFL++ to try and promote more efficient bug detection, especially in multithreaded programs.

EBF uses Bounded Model Checking, a Model Checking technique to scan for different specified properties in order to find more effective testcases for AFL++ to start off with. Furthermore, it includes its own additional instrumentation which injects pauses in execution of different threads in order to increase the likelihood of concurrency issues such as deadlocks or data races occurring in multithreaded programs, which then get caught by the Sanitiser (Clang TSAN/ThreadSanitizer). Although the MQTT brokers tested in this project will be mostly singlethreaded implementations, the use of Bounded Model Checking to generate initial inputs may allow for a significant speedup in vulnerability discovery compared to the use of a sample of regular inputs. One of out ultimate aims of the project is to serve as an addition to the whole EBF tool for use in testing programs that accept inputs in the form of MQTT packets.

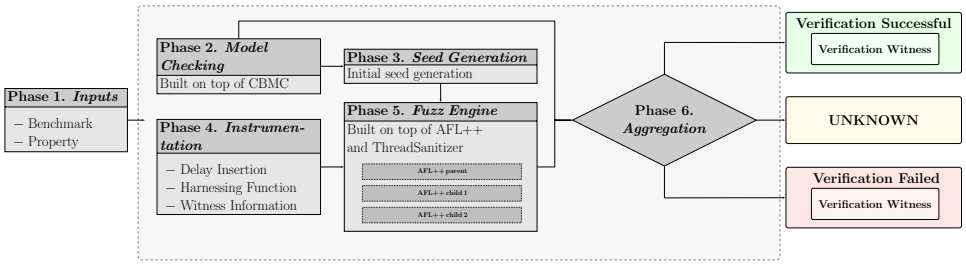

Figure 4: The EBF framework. [1]

2.7 Other Relevant Fuzzers

One of the most significant current contributions to MQTT fuzzing has been MultiFuzz [52] - a fuzzer that combines a new packet-aware mutation strategy and a fast desocketing tool in order to fuzz publish/subscribe protocols such as MQTT. MultiFuzz splits testcases into packets, and extracts random packets from previous testcases to add to new testcases, rather than just randomising the data input. It has proven to be far more effective than AFL alone, or any standard fuzzer not tailored to pub/sub protocols, discovering a lot more paths in a shorter amount of time, as it retains the format of packets, such that if a program reaches a series of states after a combination of packets, this series of states can be reproduced with high likelihood by the fuzzer.

However, although MultiFuzz has been specifically tailored to pub/sub protocols, it does not implement MQTT packet grammar as intended in our project, and thus further improvements can be made. It has to be noted that the desocketing tool developed alongside 'MultiFuzz' has proven to be extremely useful in simulating multi-client connections, and thanks to it being open-source, will also be used in the fuzzing environment in this project.

A grammar-based MQTT fuzzer, MQTTGRAM [3], has also been developed, however, does not utilise a state-of-the-art fuzzing solution such as AFL. Furthermore, due to its closed-source nature and lack of comparison to other state-of-the-art solutions, its effectiveness cannot be measured compared to other available tools. The same notion applies for other closed-source solutions such as [7] and [43].

A number of dumb fuzzers with MQTT grammar implemented are also available, most notable F-Secure's 'mqttfuzz'[12], with which comparisons will be made in the experimentation phase of the project. With consideration of the lack of mutation or program analysis of any sort, we can expect mqttfuzz to not reach the level of effectiveness seen in other previously mentioned solutions.

3 Design and Implementation

3.1 Project Methodology

The main drawback of the MultiFuzz fuzzer was the inability to implement MQTT packet grammar in the fuzzing stage, leading to many supposedly malformed packets being sent. The idea is to create a custom mutator module which creates mutated packets that can still be accepted as not malformed by MQTT brokers. In order to this, we will create a custom mutator module in AFL++, which will be run instead of the standard mutator. This custom mutator will follow the OASIS 3.1.1. MQTT Specification [29] in constructing MQTT packets. The solution will first be run with AFL++ alone, and we aim to later integrate it with the EBF [1] tool to give us higher quality initial seeds. We will start with a set of initial testcases tailored to the packet format set by the custom mutator, with the custom mutator generating new testcases based on these, as well as completely new testcases following MQTT grammar.

3.2 MQTT Packet Structure

3.2.1 Packet Types, Typical Communication and Grammar

Though briefly described in section 2.2, in order to provide a more detailed insight into the way the packets are constructed, its' important to look at the main types of packets that a client can send to an MQTT broker. These are as follows: [29]

Packet Type	Header	Description
CONNECT	0x10	Connect to Broker and begin session
PUBLISH	0x30 (+[0-255])	Publish data with given topic
SUBSCRIBE	0x81	Subscribe to Topic
UNSUBSCRIBE	0xA1	Unsubscribe To Topic
PUBREC	0x50	Acknowledge PUBLISH Received (QoS 2)
PINGREQ	0xC0	Ping MQTT Broker to check connection
DISCONNECT	0x81	End Session and Disconnect from Broker

In practice, if Transport Layer Security (TLS) is used as an additional means of security, then AUTH packets are also sent both from client to broker and broker to client in order to provide authentication between the two parties and enable encrypted packets to be sent back and forth. Fuzzing encrypted client-broker communication requires the MQTT broker to be run in a custom configuration, and public\private key exchange to be established between client and broker, which is beyond the scope of this project.

Two other packet types, PUBACK and PUBREL, can also theoretically be sent by the client to the broker, however, in typical exchanges are often sent from server to client, and are thus set aside as further development goals of the project. Different types of packets are sent back from broker to client, such as CONNACK and SUBACK, however, those do not need to be considered since they are not accepted in client-to-broker communication.

Any exchange between client and broker starts with a CONNECT packet, and continues with an exchange of multiple different packets sent by the client. Therefore, there is little use in sending singular packets to the broker, since this will only trigger either of two behaviours:

1. If CONNECT packet is properly constructed - A connection acknowledgement from the broker followed by session closure
2. Otherwise - A connection closure due to a protocol error

Therefore, apart from the odd edge case, a sequence of packets starting (with a CONNECT packet) will be sent to the broker rather than only singular packets - this will be done with the help of the 'desockmulti' tool, further details of which are highlighted in 4.3.1. A typical exchange follows the client-server request-response structure: [41]

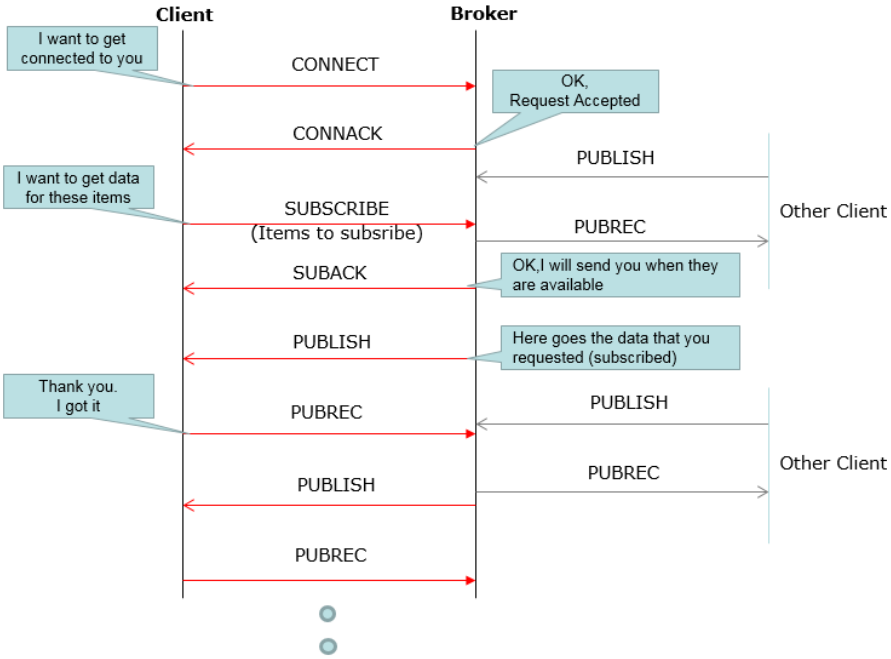

Figure 5: Typical client-broker communication in the MQTT Protocol [41]

.

3.2.2 MQTT Packet Format

Due to the lightweight nature of the protocol, all packets are sent in binary format with the following structure:

- **Fixed Header** (Always present) - Of a size of at least 2 bytes and present in every MQTT packet, this consists of the following:
  - **Control Header** - A 1-byte value, in which the most significant (leftmost) 4 bits represent the packet type, and the least significant (rightmost) 4 bits represent additional controls such as QoS level.
  - **Remaining Length** - A 1-4 byte value specifying the remaining length of the packet not including the fixed header (size of variable header + payload)
- **Variable Header** (Not always present) - Some packet types such as PUBLISH and SUBSCRIBE will include a variable header, in which information such as the packet identifier will be included, or in the case of PUBLISH, the topics specified.

- **Payload** (Not always present) - This is where additional packet-specific data is added, such as application data in PUBLISH packets, as well as topic names in SUBSCRIBE packets.

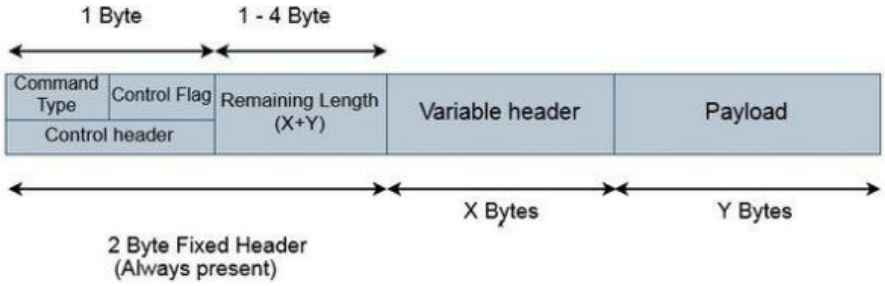

Figure 6: The format of MQTT packets

Due to their binary nature, packets are first declared as a struct in C for easier navigation and maintainability, and are serialized into a buffer once construction is complete. Due to difficult maintainability with a variable remaining length, MQTT packet deserialization is seen as relatively inefficient, will not take place in our project. Once a packet is constructed by the custom mutator, its contents will not be altered.

Each type of MQTT packet follows its own set of rules for the presence of a variable header and payload (as well as additional controls) and their contents, and can be summarised in the following table:

Packet Type	Variable Header	Payload
CONNECT	Protocol Name and Connection Configuration	Protocol Username\Password and Will Topic\Length
PUBLISH	Topic Name and Packet Identifier	Application Data
SUBSCRIBE	Packet Identifier	List of Topics and QoS
UNSUBSCRIBE	Packet Identifier	List of Topics
PUBREC	Packet Identifier	N/A
PINGREQ	N/A	N/A
DISCONNECT	N/A	N/A

3.2.3 Further Packet Grammar

In addition to following a strict packet format, further rules are set out in the specification regarding the construction of packets:

Remaining Length Calculation - Remaining length in packets is calculated in a somewhat unorthodox manner, with the size of the remaining length section changing depending on its value. The length starts at the 2nd byte from left to right, while the most significant byte is used as a flag to determine whether the next byte still represents the remaining length (0 if next byte isn't remaining length field, 1 otherwise). As an example, the largest 1-byte remaining length is 127, represented by 01111111, with the next remaining length, 128, being represented as 10000000 00000001, and 129 being 10000000 00000010. Conveniently the OASIS MQTT 3.1.1 Specification [29] includes a rough C pseudocode for encoding the remaining length, which has been utilised in this project.

CONNECT Packet Grammar - Further highlighted in the specification, certain connection configurations should not be allowed by the broker, such as having a password flag without a username flag. Furthermore, the contents of the payload in the CONNECT packet depend heavily on the contents of the variable header (e.g. Username flag in variable header signifies the presence of a username field in the payload). Due to the whole client-broker connection being reliant on a CONNECT packet being accepted, all rules laid out in the specification [29] have been implemented in the custom mutator's packet construction procedures.

QoS Level and Associated Packets - In addition to regular MQTT communication described previously, the MQTT specification also allows for additional reliability checks in the form of Quality of Service Levels 1 and 2. When QoS levels 1 or 2 are selected, communication involves additional acknowledgement packets to make sure transmission has taken place properly. Due to the highly complex nature of QoS 1 and 2 communication in relation to regular QoS Level 0 communcation in MQTT, no QoS level grammar has been implemented (with the exception of the CONNECT packet, for which it is vital to be constructed accurately), and instead, all packets involving specification of a QoS Level have had the QoS level set to a random value from 0 to 2.

3.3 Fuzzing Environment

3.3.1 Desockmulti

The first challenge that took place in our project was fuzzing a program that accepted inputs through a network socket rather than through standard input, since AFL++ sent all its inputs through standard input. In order to work around this, there were two different solutions:

- Use a hooking tool such as Preeny's desock() [50] to hook socket functions and hijack the broker's socket functions to accept inputs from standard input. The 'desocketing' tool is then preloaded whenever AFL++ is run
- Configure AFL++ to send packets directly through ordinary sockets into the broker

The latter solution would require modification of the AFL++ source code, and AFL documentation recommends the use of a desocketing tool, so this is the solution that we also utilise in our project. As previously mentioned in 3.2, sending singular packets to the MQTT broker will render limited results, and thus we need to be able to simulate a full session consisting of multiple packets being sent from client to broker. Furthermore, typical MQTT communication involves multiple parties (more than 2) rather than standard two-party communication, as we have the 'broker', 'publishers' and 'subscribers' (see 2.2). Therefore, in order to provide a more realistic testing environment, we need to be able to simulate multiple clients interacting with the MQTT broker, rather than just a single client.

In order acheive this, we use the open-source 'desockmulti' [53] desocketing tool provided by the creators of the MultiFuzz [52] fuzzer. Desockmulti, similarly to Preeny, is preloaded when AFL++ is run (either through LD_PRELOAD or AFL_PRELOAD arguments specified when running AFL++) and hooks the basic socket(), bind(), listen() and accept() functions of a network program. Unlike Preeny, however, it presents the following advantages:

- Multiple connection support (i.e. simulation of multiple clients in a single fuzzing instance), with multiple packets being able to be sent through a single connection.
- Optimisation for fuzzing, providing a 10x speedup over standard desock().

For these reasons, we have tailored the custom mutator to work in conjunction with the desockmulti tool rather than constructing standard MQTT packets sent over the desock tool.

Desockmulti seed format - Due to the additional features provided by desockmulti, it requires the use of a custom seed format that is wrapped around MQTT packets. Each seed starts with a 2-byte header. The first byte (Accept Number) specifies the number of sockets that connnect to the broker's accepting socket. The second byte (Connect Number) provides the number of sockets to which the broker connects to, which in this case will be set to 0 at all times, since the MQTT brokers tested in our project do not initiate any connections. Following these two bytes, we have desockmulti messages, which consist of: [52]

- **Socket Index** - A 1-byte value specifying the index to which the message belongs to
- **Message Length** - A 2-byte value specifying the length of the message content
- **Message Content** - The message being sent, in our case, the serialized MQTT packet itself, the size of which is specified by the message length.

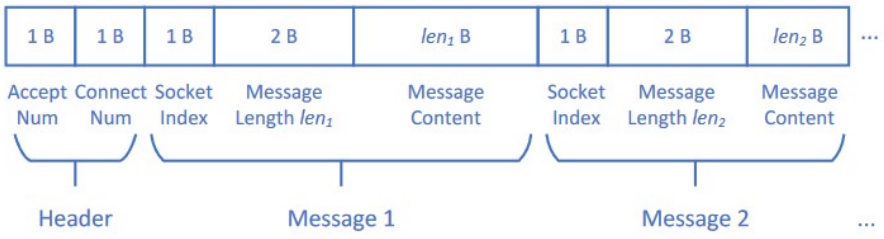

Figure 7: The Desockmulti packet format. [52]

Using the example provided in the MultiFuzz paper, we see a packet with Accept Number 2 and Connect Number 0, meaning that two clients will be simulated. The first packet will be sent by the first client (Socket Index 00) with a length of 4 bytes and the message '00 11 22 33', and the second packet will be sent by the second client (Socket Index 01) with a length of 2 bytes, and the message 'FF EE'. Though in our example, these messages are random values, these will be of the standard MQTT packet format when running the program itself.

Figure 8: An Example Desockmulti seed [52]

Due to the message length being limited to 2 bytes, only MQTT packets of a maximum size of 65535 bytes can be sent to the broker by desockmulti - a size significantly smaller than the maximum accepted size of 268435455 bytes provided by the OASIS specification. With this in mind, it is unlikely that our current fuzzing environment will be able to discover any overflow-related vulnerabilities (e.g buffer overflow) in the MQTT brokers tested - this is a point highlighted in the project's further improvements.

3.3.2 AFL++ Environment

To summarise, we now use a modified environment of the standard AFL\AFL++ Fuzzing setup, which desockmulti acting as an intermediary between afl-fuzz and the PUT. Desockmulti seeds containing several packets are fed into desockmulti, which in turn creates the specified number of sockets and feeds packets through these sockets accordingly. The custom mutator creates new seeds of the desockmulti format, with embedded packets following the MQTT packet grammar accordingly. The desockmulti seed generation strategy will be outlined in the next subsection.

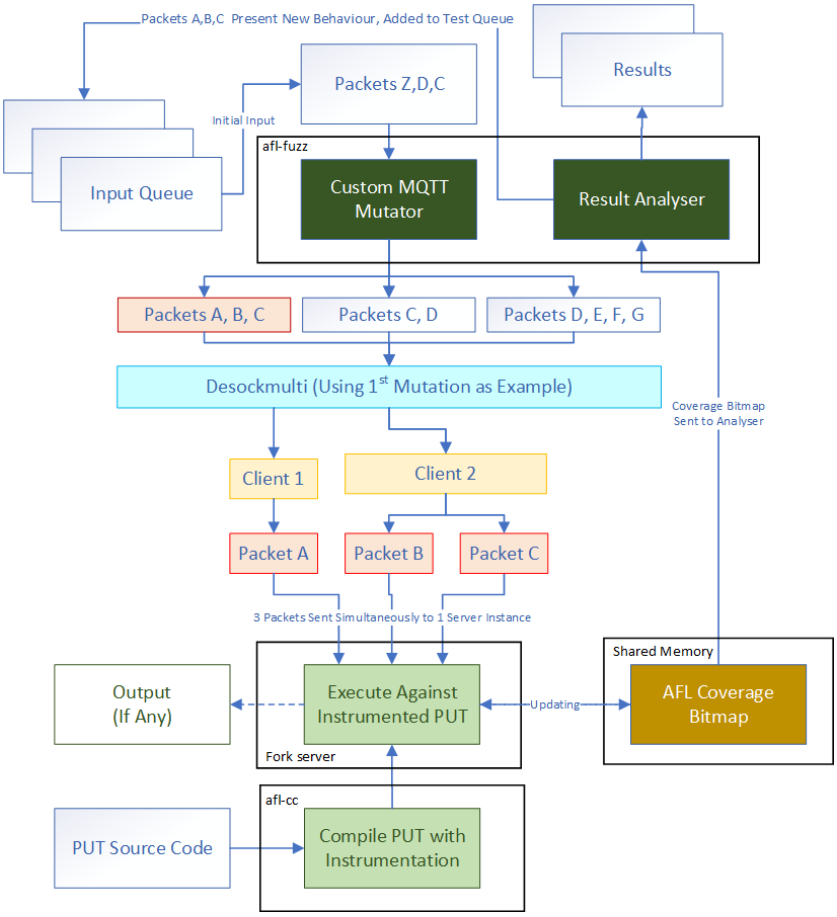

Figure 9: Our updated fuzzing environment utilising the 'desockmulti' tool

3.4 Mutation Strategy

In our custom mutation strategy, we provide a packet-aware strategy similar to the implementation seen in MultiFuzz [52] to alternate between packets and retain the desockmulti seed structre. The strategy is outlined as follows:

- **Create Entirely New Desockmulti Seed** (40% Chance)
- **Add a packet Onto The Current Desockmulti Seed** (40% Chance)
  - *Select from Existing Queue* (20% Chance)
  - *Create New Packet* (20% Chance)
- **Remove A Random Packet from Current Desockmulti Seed** (15% Chance)
- **Chance Desockmulti Connect or Accept Number** (5% Chance)

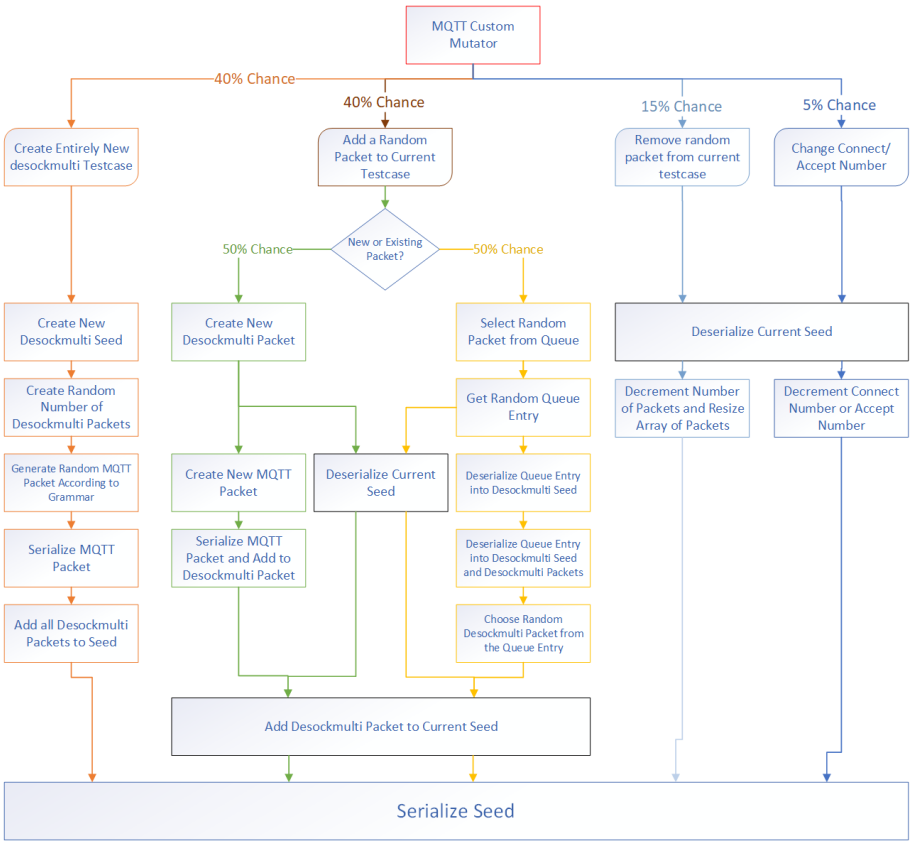

Figure 10: An Outline of Our Mutation Strategy and the Steps that Entail It

Note: In this chapter, the phrases 'seed' and 'testcase' are used interchangably and are equivalent.

3.4.1 Create Entirely New Seed

When this process is selected, we first randomise the amount of packets in the desockmulti seed. We then create `desockmulti_testcase()` struct pointer and allocate the appropriate amount of memory to it depending on the amount of packets we have randomised. The desockmulti packet is then populated using `dsm_packet_create`. We start by randomising the accept number, and setting the connect number to 0. Every time a desockmulti packet is created, there is a 40% chance that the socket index of the packet will be incremented from the previous value, with the maximum socket index being the one less than the randomised accept number. For every creation of a desockmulti packet, the socket index is first set, and then the mqtt packet is formed by `packet_create` and then serialized into a buffer of appropriate size, a pointer to which is stored in the desockmulti packet. The `serialize_mqtt_packet` function returns a value for the size of the mqtt packet (including the control flag) in bytes, which is later set as the message length value in the desockmulti packet. A pointer to the buffer containing the serialised MQTT packet is then set as the `mqtt_packet` pointer of the desockmulti packet. A pointer to the constructed desockmulti packet is stored in the `packet_array` pointer array of the `desockmulti_testcase`. Once the entire `desockmulti_testcase` is populated, it is then serialized into the output buffer.

3.4.2 Add A Packet Onto the Current Desockmulti Seed

When this process is selected, the current testcase input is first deserialized back into a `desockmulti_testcase()` struct, with the appropriate amount of memory assigned to accomodate a new packet. The following two actions can then be taken:

- A new desockmulti packet is created and added to the current desockmulti seed
- A random existing desockmulti packet is extracted from a random testcase in the corpus and added to the current desockmulti seed

If a new desockmulti packet is created, then the creation process follows the same methodology as seen in creating an entirely new seed.

The extraction of a current testcase follows a different set of procedures. First, a similar procedure as the one seen in MultiFuzz [52] has been used to find a random queue entry and map its contents into a buffer of appropriate size using C's `mmap` function - this buffer is then serialized into a `desockmulti_testcase()` struct, with the individual desockmulti packets also being serialized in the process. A random `desockmulti_packet` pointer is then chosen from the `packet_array` of the `desockmulti_testcase` struct.

In both cases, once the pointer to the appropriate `desockmulti_packet` is chosen, the a new `desockmulti_testcase` is created with the same data as the current seed, but with larger memory to account for the additional packet pointer. The new packet is then added to the end of the `packet_array` of the new seed, and the testcase is serialized into the output buffer.

3.4.3 Remove Random Packet from Current Seed

When this process is selected, the current seed is deserialized into a `desockmulti_testcase` struct, and a random pointer is chosen from the packet. The pointer is swapped with the last pointer in the array and then cleared. C's `realloc` function is used to resize the testcase into the appropriate smaller size, and the new testcase is written into the output buffer.

3.4.4 Change Desockmulti Accept/Connect Number

A strategy also seen in MultiFuzz [52], the current testcase is deserialized into a `desockmulti_testcase` struct, one of either the Accept number of Connect Number (50% chance) is either incremented or decremented (50% chance) and the `desockmulti_testcase` struct is written back into the output buffer.

3.5 Side Endeavor: Vulnerability Research in MQTT Clients

Throughout the initial research stage of the project, various other ideas related to fuzzing were experimented with. In this, an attempt was made at vulnerability discovery in MQTT client implementations using different instrumentation tools such as AddressSanitizer [39] and ThreadSanitizer [40]. Throughout this research, we managed to find a data race vulnerability in the multithreaded example client in the WolfMQTT [47] Library thanks to compilation using ThreadSanitizer. Following the presentation of this project, the vulnerability will be responsibly disclosed to the WolfSSL developer team.

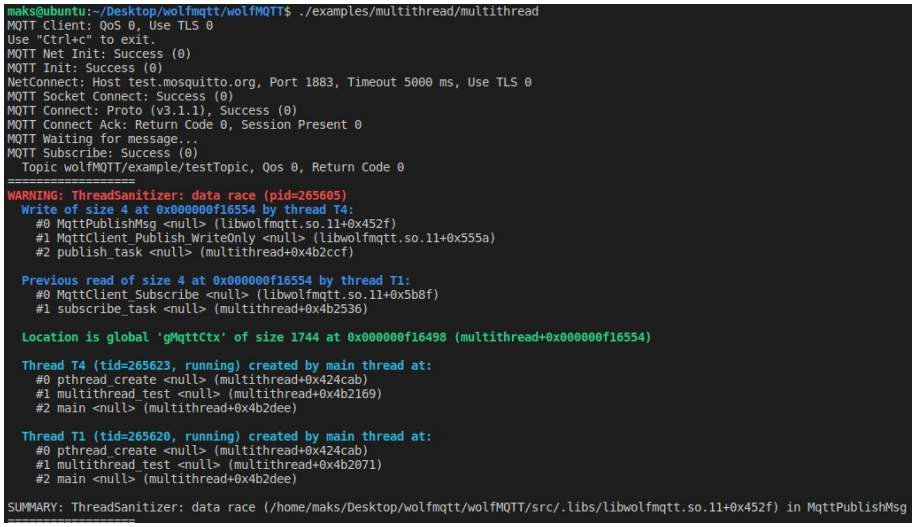

Figure 11: The Data Race Vulnerability found in the Multithreaded WolfMQTT Client

4 Evaluation

4.1 Evaluation of Mutation Strategies

In our project, we have developed a set of improvements that give the AFL++ both packet awareness and grammar awareness in fuzzing MQTT brokers, and can be compared to existing MQTT fuzzing solutions below:

Fuzzer	Fuzzer Type	Grammar-Aware?	Packet-Aware?
mqtt_fuzz	Dumb	No	No
AFL++	Coverage-Based	No	No
MQTTGRAM	Grammar-Based	Yes	No
MultiFuzz	Coverage-Based	No	Yes
AFL++ with Our Mutator	Coverage-Based	Yes	Yes

Our solution ultimately combines a packet-aware fuzzing strategy similar to the one seen in MultiFuzz [52] with the MQTT grammar seen in MQTTGRAM to create a new and improved fuzzing strategy atop a state-of-the-art fuzzing solution in the form of AFL++.

4.2 Testing on MQTT Broker Implementations

4.2.1 Testing Environment

In order to measure the effectiveness of our new mutation strategy, two popular MQTT brokers have been selected for testing - Eclipse Mosquitto [25] and NanoMQ [11].

Eclipse Mosquitto - We used Mosquitto version 2.0.14 for testing, which implements MQTT Protocol versions 5.0, 3.1.1 and 3.1.

NanoMQ - We used NanoMQ version 0.7.5 for testing, which also implements MQTT Protocol versions 5.0, 3.1.1 and 3.1.

Testing has been conducted on a VMWare Workstation 15 Virtual Machine with 10GB of allocated RAM and 2 CPU Cores allocated from the parent CPU - an Intel Core i7-6700HQ processor. Across the first hour of testing, we took results per 5 minutes and averaged them out for each fuzzing solution. Testing for longer than 1 hour did not take place.

4.2.2 Testing Results

In our testing stage, we ran 3 different solutions - AFL++, MultiFuzz and AFL++ with our Custom Mutator Module over a set of different short time frames, with the averages being calculated below:

Solution	Avg Execution Speed	Avg Total Paths/5min	Paths/1k Executions
Standard AFL++	472 execs/s	16	0.488
Custom Mutator	146 execs/s	41	1.250
MultiFuzz	620 execs/s	285	1.357

Furthermore, in one of the runs of our custom mutator, a crash was found in the Mosquitto broker application:

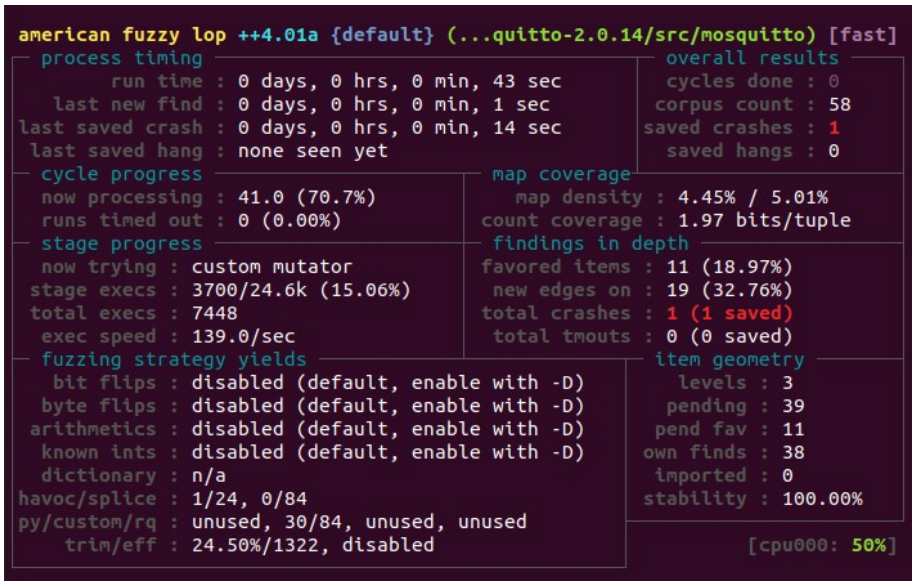

Figure 12: A crash that was found in Mosquitto 2.0.14 having only run our custom mutator for 48 seconds

4.3 Evaluation of Results

All of our results were taken across the 1st hour of fuzzing the brokers using the virtual machine, as the number of new paths discovered reaches a plateau upon more hours of fuzzing, and as such, the 1st hour of fuzzing is rendered to be most significant. Our custom mutator proved to be more effective at finding new paths in the MQTT broker code compared to a standard state-of-the-art AFL++ generic solution, but fell significantly short of MultiFuzz in path discovery per 5 min. The amount of paths discovered per 1000 executions, however, are fairly similar, signifying this is most likely an issue of our custom mutator's slow execution speed, with MultiFuzz achieving over 4x the amount of executions per second. Inefficient memory management in the form of multiple `malloc` and `realloc` calls rather than reusing buffers is likely the main cause of this, thus one of the key improvements outside of the ones highlighted in the conclusion, is to implement said reusable buffers.

Unlike MultiFuzz, however, one of the runs of our custom mutator module has managed to produce a crash of the Mosquitto 2.0.14 broker - a goal that was specifically mentioned not to have been achieved in [52]. It remains to show that this crash is reproducible (or whether it could simply be an anomaly in our testing environment) and could potentially lead to a vulnerability - an area of further investigation in the future.

Finally, in the earlier stages of our research we have discovered a data race bug (thanks to compilation with Clang ThreadSanitizer) in the WolfMQTT version 1.12.0 example multithreaded client. At the time of writing this report, WolfMQTT version 1.13.0 has been released, and it remains to show whether this bug is still present in the latest version. If so, responsible disclosure to the developer team will take place in hopes of this bug being fixed.

4.4 Threats to Validity

One of the major uncertainties throughout the development of the project was whether the testcases have been generated as intended. Efforts have been made in debugging in the form of writing packets to standalone files to be run with desockmulti against the broker without the use of AFL++ to check the validity of testcases generated, but without analysing packets bit-by-bit (a taxing process for which the time constraints of the project didn't allow for), we cannot be certain that the MQTT packets have been generated exactly as intended. Furthermore, only MQTT brokers written in C (Mosquitto, NanoMQ) have been tested, and it remains to show the effectiveness of the tool in fuzzing other MQTT brokers written in different languages, such as EMQ X and FlashMQ.

All fuzzing programs were run in the space of an hour or less, with 5-minute results being taken - this is due to the initial amount of paths in fuzzing spiking, and later reaching a plateau as the program is run for several hours, thus the most significant change in amount of paths produced takes place in the first hour. Running the experimentation for a prolonged period of time (e.g. more than 6 hours) is likely to produce more accurate results and is an area for further investigation.

Another feature to highlight in our implementation is the use of the time as a random seed in initiating our fuzzing rather than a fixed seed, and as such results generated in this experimentation phase are unlikely to be precisely reproducible in any other environment, or even in another run. The use of a fixed seed in a later version of our application will allow for reproducible results.

Finally, it remains to show that the same results can be achieved while running the program on any system, and thus a further step in proving the validity of the results would be to re-run the experimentation on a fresh, isolated system (e.g. a new virtual machine) and compare results.

5 Conclusion

5.1 Project Summary

This project has allowed for the development of a new fuzzing strategy tailored specifically towards fuzzing MQTT Brokers. There remain gaps in our solution, as execution speed is relatively low compared to all other implementations, and thus before public release, more efficient memory management needs to be implemented. Our first objective has only partially been fulfilled in the form of outperforming a generic state-of-the-art fuzzer (AFL++), rather than a more specialised solution (MultiFuzz).

Although the project experimentation has proven that MultiFuzz is still optimal for MQTT fuzzing, our novel solution has allowed for the discovery of a crash in the Mosquitto MQTT Broker - a feat not achieved by MultiFuzz. Moreover, MQTT client testing during our initial research phase has allowed for the discovery of a new data race bug in a popular MQTT client application, thus again fulfilling our second objective of finding vulnerabilities in applications that utilise the MQTT Protocol. With this in mind, it is safe to assume that the second objective highlighted in our report has been achieved.

A workable custom mutator module solution used in this project will be made available alongside all additional programs necessary (e.g. AFL++, desockmulti) on my personal GitHub page: `https://github.com/mgotko`

5.2 Further Improvements

5.2.1 Most Significant Improvements

A variety of further improvements can be made in order to generate more effective results. The following two improvements would prove to be the most academically significant and would be most likely to discover new vulnerabilities:

- **Support for AUTH Packets** - The next stage to develop is fuzzing MQTT communication via TLS, as a lot of sensitive information uses this format rather than a standard MQTT communication channel. In order to achieve this, a number of modifications would have to be made. The most significant of which, would be the establishment of a public\private key scheme between broker and client, and an encryption function which uses the broker's public key to encrypt packets before sending them. Other modifications include changing the broker's configuration to TLS communication, and implementing the grammar for AUTH packets being sent.
- **Generation of packets with size larger than 65KB** - A major limitation of the desockmulti tool is the 2-byte message length value in each desockmulti packet, which only allows for a maximum packet size of 65535 bytes. As previously mentioned, the limited packet size will not allow for generation of edge case packets, in which a large size could trigger a buffer overflow in the code. A change from a 2-byte message length value to a 4-byte message length would allow for packets to be created close (and potentially over) the 268435455-byte limit highlighted in the OASIS MQTT Specification [29], and thus could potentially trigger new unexpected behaviours, and the possibility of new vulnerabilities being discovered.

5.2.2 Other Potential Improvements

Other improvement ideas that have the potential positive impact on fuzzing results and path discovery are the following:

- **Optimisations to Current Mutation Strategy** - At the moment, the custom mutator module utilises a lot of `malloc` statements upon packet creation and packet extraction, leading to reduced efficiency - replacing these memory allocations with reusable buffers (e.g. would allow for a significant speedup to the module overall.
- **Implementation of Proper QoS Grammar** - Support for PUBACK, PUBREL, and PUBCOMP packets being sent from client to broker as well as the implementation of stricter QoS grammar (e.g. tracking which packets have been sent with which QoS level) could allow for new paths relating to the handling of QoS-related messages being discovered. While generation of random PUBACK, PUBREL and PUBCOMP packets is a fairly trivial process and one of the most likely next steps in developing the custom mutator, the implementation of a stricter QoS will require a larger overhead and will provide an additional layer of compexity to the mutation strategy. For this reason, a stricter QoS grammar could prove to provide more drawbacks in both fuzzing efficiency and the ease of use of the custom mutator relative to any improvements in path discovery that it would provide.
- **A Custom Havoc Mutation Procedure** - At the moment, the custom mutator uses a the standard `surgical_havoc_mutate` procedure provided in the custom mutator helper function header in order to generate random bit values for MQTT packet contents. The MQTT specification [29], however, states that in most cases packet contents should be of the UTF-8 format, a subset of the set of all possible bit values. This likely means that a lot of packets are potentially being rejected because the contents are not entirely of the UTF-8 format - a custom havoc mutation function that only sets bits

to UTF-8 values has the potential of fixing this. On the other hand, limiting packet contents to only UTF-8 values has the potential of missing a lot of paths, and therefore it is likely that the utilisation of both a custom havoc mutation procedure and the standard procedure provided by AFL++ (e.g 50% chance of custom procedure, 50% chance of generic procedure in order to generate packet contents), would be the best compromise for effective path discovery.

- **Support for the EBF Tool** - Although initially a significant goal of the project was to provide a direct improvement to the EBF tool as a whole, the integration of the improved custom mutator into the EBF tool proved to be beyond the scope of the project. The use of model checking in generation of initial inputs could potentially allow for more efficient fuzzing, however, several hurdles will have to be overcome. Any initial inputs that the EBF model checker provides would then have to be wrapped into desockmulti testcases before being input into the fuzzer, and it would have to be made sure that these initial inputs somewhat follow MQTT grammar (with the understanding that malformed packets could also trigger interesting behaviours) in order to not risk being immediately rejected by the broker.

- **Addition of Random Havoc Mutations** - One of the simpler potential improvements, an additional mutation strategy could be added in form of simply adding a random standard AFL++ havoc mutation (bit flip, word addition, etc.) to the current testcase in hopes of triggering new behaviours. This random mutation is likely to result in either desockmulti or the MQTT broker simply rejecting the packets, and thus, if implemented, should be set to run at a low probability of occurrence (e.g. 5% Chance of using this strategy).

- **Probability of Incorrect MQTT Packet grammar** - Another interesting strategy was presented by Rodriguez and Batista in the MQTTGRAM tool [37], in which the fuzzer ran with a probabilty of packets being generated in the incorrect order. A similar idea implemented in our custom mutator could potentially allow for new path discovery, however, a large number of packets sent in the incorrect order would decrease the efficiency of the fuzzer.

References

[1] Fatimah Aljaafari, Lucas C Cordeiro, Mustafa A Mustafa, and Rafael Menezes. Ebf: A hybrid verification tool for finding software vulnerabilities in iot cryptographic protocols. *arXiv preprint arXiv:2103.11363*, 2021.

[2] Kaled M Alshmrany, Rafael S Menezes, Mikhail R Gadelha, and Lucas C Cordeiro. Fusebmc: A white-box fuzzer for finding security vulnerabilities in c programs (competition contribution). *Fundamental Approaches to Software Engineering*, 12649:363, 2021.

[3] Luis Gustavo Araujo Rodriguez and Daniel Macêdo Batista. Program-aware fuzzing for mqtt applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 582–586, 2020.

[4] Kevin Ashton et al. That 'internet of things' thing. *RFID journal*, 22(7):97–114, 2009.

[5] Harald Bauer, Mark Patel, and Jan Veira. The internet of things: Sizing up the opportunity. *Retrieved from: McKinsey at http://www.mckinsey. com/insights/high_tech_telecoms_internet/the_internet_of_things_sizing_up_the_opportunity*, 2014.

[6] Sergio Campos and Edmund Clarke. *Model Checking: An Overview*. Federal University of Minas Gerais, 2010. URL: `https://homepages.dcc.ufmg.br/~scampos/cursos/rt/aulas/verificacao03.pdf`.

[7] G Casteur, A Aubaret, B Blondeau, V Clouet, A Quemat, V Pical, and Rafik Zitouni. Fuzzing attacks for vulnerability discovery within mqtt protocol. In *2020 International Wireless Communications and Mobile Computing (IWCMC)*, pages 420–425. IEEE, 2020.

[8] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. Muzz: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2325–2342, 2020.

[9] CVEDetails. Cvedetails - eclipse mosquitto : Security vulnerabilities, 2022. URL: `https://www.cvedetails.com/vulnerability-list/vendor_id-10410/product_id-45945/Eclipse-Mosquitto.html`.

[10] Callum Cyrus. Iot cyberattacks escalate in 2021, according to kaspersky, Sep 2021. URL: `https://www.iotworldtoday.com/2021/09/17/iot-cyberattacks-escalate-in-2021-according-to-kaspersky/`.

[11] EMQX. Nanomq. URL: `https://github.com/emqx/nanomq`.

[12] F-Secure. F-secure/mqt_fuzz: A simple fuzzer for the mqtt protocol. URL: `https://github.com/F-Secure/mqtt\textunderscorefuzz`.

[13] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[14] FitBit. Fitbit - innovation meets motivation. URL: `https://www.fitbit.com/global/uk/home`.

[15] Eclipse Foundation. Eclipse foundation iot and edge developer survey report, 2021. URL: `https://outreach.eclipse.foundation/iot-edge-developer-2021`.

[16] Mikhail R Gadelha, Felipe Monteiro, Lucas Cordeiro, and Denis Nicole. Esbmc v6. 0: verifying c programs using k-induction and invariant inference. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 209–213. Springer, 2019.

[17] Christopher Guindon. Eclipse desktop & web ides: The eclipse foundation. URL: `https://www.eclipse.org/ide/`.

[18] Hive. Hive home us: Start your connected home. URL: `https://www.hivehome.com/`.

[19] Urs Hunkeler, Hong Linh Truong, and Andy Stanford-Clark. Mqtt-s—a publish/subscribe protocol for wireless sensor networks. In *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE'08)*, pages 791–798. IEEE, 2008.

[20] @keymandll. Improper handling of payload length #203. URL: `https://github.com/emqx/nanomq/issues/203`.

[21] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[22] Selena Larson. Fda confirms that st. jude's cardiac devices can be hacked, Jan 2017. URL: `https://money.cnn.com/2017/01/09/technology/fda-st-jude-cardiac-hack/`.

[23] Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD conference*, volume 5, 2008.

[24] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):1–13, 2018.

[25] Roger A Light. Mosquitto: server and client implementation of the mqtt protocol. *Journal of Open Source Software*, 2(13), 2017.

[26] Microsoft. Visual studio code - code editing. redefined, Nov 2021. URL: `https://code.visualstudio.com/`.

[27] Charlie Miller. How smart is intelligent fuzzing-or-how stupid is dumb fuzzing. *Independent Security Evaluators*, 2007.

[28] Jayashree Mohanty, Sushree Mishra, Sibani Patra, Bibudhendu Pati, and Chhabi Rani Panigrahi. Iot security, challenges, and solutions: a review. *Progress in Advanced Computing and Intelligent Engineering*, pages 493–504, 2021.

[29] OASIS. Mqtt version 3.1.1, 2014. URL: `https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html`.

[30] OASIS. Mqtt version 3.1.1, 2018. URL: `https://blog.axway.com/apis/api-builder-and-mqtt-for-iot-part-1`.

[31] US Bureau of Labor Statistics. Producer price index by industry: Semiconductor and other electronic component manufacturing, Apr 2022. URL: `https://fred.stlouisfed.org/series/PCU33443344`.

[32] Sumit Padhiyar and KC Sivaramakrishnan. Confuzz: Coverage-guided property fuzzing for event-driven programs. In *International Symposium on Practical Aspects of Declarative Languages*, pages 127–144. Springer, 2021.

[33] GitHub Project. URL: `https://codeql.github.com/`.

[34] GNU Project. Gcc, the gnu compiler collection. URL: `https://gcc.gnu.org/`.

[35] LLVM Project. Clang: A c language family frontend for llvm. URL: `https://clang.llvm.org/`.

[36] LLVM Project. Libfuzzer – a library for coverage-guided fuzz testing. URL: `https://llvm.org/docs/LibFuzzer.html`.

[37] Luis Gustavo Araujo Rodriguez and Daniel Macêdo Batista. Towards improving fuzzer efficiency for the mqtt protocol. In *2021 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–7. IEEE, 2021.

[38] Selenium. URL: `https://www.selenium.dev/`.

[39] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.

[40] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*, pages 62–71, 2009.

[41] ShareTechNote. Ltn (low throughput network). URL: `https://www.sharetechnote.com/html/IoT/App_Protocol_MQTT.html`.

[42] Satyajit Sinha. State of iot 2021: Number of connected iot devices growing 9% to 12.3 billion globally, cellular iot now surpassing 2 billion, Dec 2021. URL: `https://iot-analytics.com/number-connected-iot-devices/`.

[43] Hannes Sochor, Flavio Ferrarotti, and Rudolf Ramler. Automated security test generation for mqtt using attack patterns. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, pages 1–9, 2020.

[44] SonarQube. Code quality and code security. URL: `https://www.sonarqube.org/`.

[45] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[46] Yan Wang, Peng Jia, Luping Liu, Cheng Huang, and Zhonglin Liu. A systematic review of fuzzing based on machine learning techniques. *PloS one*, 15(8):e0237749, 2020.

[47] WolfSSL. Wolfmqtt: A small, fast, portable mqtt client implementation, including support for tls 1.3, 2022. URL: `https://github.com/wolfSSL/wolfMQTT`.

[48] Nicky Woolf. Ddos attack that disrupted internet was largest of its kind in history, experts say, Oct 2016. URL: `https://www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet`.

[49] Michal Zalewski. Technical whitepaper for afl-fuzz. *URl: http://lcamtuf.coredump.cx/afl/technical_details.txt*, 2014.

[50] Zardus. Zardus/preeny: Some helpful preload libraries for pwning stuff. URL: `https://github.com/zardus/preeny`.

[51] Andreas Zeller. Greybox fuzzing - the fuzzing book, 2021. URL: `https://www.fuzzingbook.org/html/GreyboxFuzzer.html`.

[52] Yingpei Zeng, Mingmin Lin, Shanqing Guo, Yanzhao Shen, Tingting Cui, Ting Wu, Qiuhua Zheng, and Qiuhua Wang. Multifuzz: A coverage-based multiparty-protocol fuzzer for iot publish/subscribe protocols. *Sensors*, 20(18):5194, 2020.

[53] Zyingp. Zyingp/desockmulti: A de-sockecting tool that is 10x faster than desock (preeny) in fuzzing network protocols. URL: `https://github.com/zyingp/desockmulti`.