Abstract

Embedded systems are prevalent in all parts of modern life, consumer electronics, home automation, and also in various safety critical areas such as medical devices or automotive systems. Their large influence means they attract malicious intent. Attackers aim to compromise data or render devices nonfunctional. Therefore, the creation of robust and secure embedded systems is paramount. Despite their ubiquity in our daily life, their financial viability has meant that in practice important aspects of embedded systems, such as security are frequently overlooked. Additionally, numerous vulnerable systems are currently available on the market without the capability for updates therefore making fixes impossible. This underscores the significance of ensuring that these systems are launched with minimal bugs and vulnerabilities.

To enhance system security, proactive vulnerability detection and mitigation are crucial. Fuzzing proves highly effective in achieving this goal.

Fuzzing is an automated software testing approach that involves introducing invalid, malformed, or unexpected inputs into a program. The program's behaviour is observed for anomalies like crashes or memory leaks, aiming to identify software defects and vulnerabilities. Despite the growing popularity of fuzzing, the field of fuzzing embedded systems lacks research publications, primarily due to the challenges associated with conducting effective fuzz testing on such systems.

In this thesis, a prototype fuzzer is created for applications that rely on SocketCAN for communication. SocketCAN is an implementation of the widely adopted Controller Area Network (CAN) protocol within embedded systems. It is a virtual machine based snapshot fuzzer addressing problematic characteristics of embedded systems like statefulness and specialized inputs. One of its primary benefits is that it emulates communication with the fuzzing target, leading to improved performance and broader applicability. This contributes to addressing the research gaps concerning security in embedded systems and the practice of fuzzing within this context.

Furthermore, this thesis includes a discourse on the subject of fuzzing applications utilising embedded protocols, aiming to encourage further exploration in this domain. This knowledge can be employed to expand the developed prototype fuzzer and to contribute to various other fuzzing projects as well.

Contents

CHAPTER 1

Introduction

In today's technologically advanced world, embedded systems play an integral role in our daily lives. The rapid growth of the Internet of Things (IoT) has greatly contributed to this situation. A recent study showed that there should exist approximately 30 billion IoT devices by 2030 [40]. They are now pervasive in various industries, including Automotive Systems, Aerospace and Aviation, and Medical Devices. The integration of embedded systems in these industries demands a robust approach to safeguard against potential vulnerabilities because they have become essential for the functioning of societies and economies, and any vulnerabilities or disruptions can have severe consequences. In reality, these systems are often insecure. The Mirai malware [12] was a notorious cyber attack that hijacked vulnerable IoT devices to create a massive network of compromised devices, enabling large-scale distributed distributed denial-of-service (DDoS) attacks. Beavers et al. [21] showcased that it is feasible to hack current models of pacemakers. In a recent incident, the Controller Area Network (CAN) bus, which is a standard component in nearly all modern vehicles, was compromised, allowing the theft of a new Toyota RAV4 [3]. Although not directly tested, it is reasonable to assume that this attack could be replicated on other automobile models as well.

Detecting and fixing vulnerabilities prior to exploitation by malicious actors can significantly enhance the security of these systems. Fuzzing presents itself as a highly effective approach in accomplishing this objective.

Fuzzing or fuzz testing is an automated software testing technique that injects invalid, malformed, or unexpected inputs into a program. The program is monitored for exceptions such as crashes or memory leaks, with the aim to detect software defects and vulnerabilities that could be exploited by a malicious attacker. By subjecting a system or application to a wide range of inputs, including malformed or unexpected data, fuzzing helps identify weak points that may have been overlooked during development or testing. Due to the limited testing depth achieved by pure random input, advanced fuzzing tools leverage supplementary information, such as input structure or code coverage, to generate inputs

with a higher likelihood of reaching substantial sections of the code base. With its ability to uncover unknown vulnerabilities and trigger unexpected software behaviours, fuzzing has become a valuable technique in improving the security and reliability of software systems.

In the past, the majority of research was done to improve the fuzzing algorithms [57]. Grammar-based and coverage-guided fuzzing were primary research domains that received significant attention. Grammar-based fuzzing involves creating a grammar that specifies the structure and syntax of the input data that the program under test can accept. Grammar-based fuzzing is useful when the input data format is complex, it leverages a grammar to generate inputs that adhere to the required structure. Coverage-guided fuzzing involves using feedback from the program under test to guide the fuzzing process. The feedback can be obtained by instrumenting the program and collecting code coverage information. Coverage information can be used to guide the generation of new inputs that are more likely to explore previously untested parts of the code. It is useful when the goal is to test as much of the target's code as possible. The idea behind snapshot-based fuzzing is to capture and reuse program states during the fuzzing process. Instead of starting each fuzzing iteration from scratch, the fuzzer captures a snapshot of the program. This approach improves efficiency and increases test case diversity. Combining the grammar-based and coverage-guided approaches with snapshot based fuzzing makes them even more efficient and enables the discovery of vulnerabilities that might otherwise remain undetected.

It is increasingly apparent that the capability to fuzz new targets is proving to have a greater practical impact as opposed to refining the algorithmic level. It is highly probable that any application subjected to fuzzing for the first time will reveal numerous security-relevant discoveries [23]. Hence, this thesis endeavors not to enhance the algorithmic level of fuzzers, but instead to leverage the present state of contemporary fuzzers, particularly coverage-guided snapshot-based, to perform fuzzing on systems utilising embedded system protocols. The significance of pursuing this approach stems from the inherent limitations in utilising existing fuzzing research for embedded systems, mainly due to obstacles such as multi-architecture support [25], difficulties in crash detection [46], and resource constraints. The absence of adequate research leaves these systems and their domains vulnerable. Additionally, the limited applicability of published embedded system fuzzing research further compounds the issue, as their usability beyond their proposed example scenarios becomes increasingly challenging.

1.1 Expected Results

The aim of this thesis is to increase the applicability of modern fuzzing techniques on systems using embedded system protocols. Since there is currently no widely applicable state-of-the-art open-source fuzzer available for such systems, the goal is achieved by developing a new prototype fuzzer. To leverage the advancements in current research, the prototype in this thesis is built upon an existing open-source fuzzer. A suitable choice

for this purpose is Nyx-Net [57], a coverage-guided, snapshot-based fuzzer designed for targeting network-based systems.

Systems using embedded protocols are often stateful and involving passing multiple messages back and forth, which poses unique challenges in contrast to traditional fuzzing targets. Nyx-Net tackles these challenges with its design principles that include hypervisor-based snapshot fuzzing to ensure noise-free fuzzing and to speed up resetting to a clean slate, and selective emulation of network functionality to avoid the heavy cost of real network traffic. Another obstacle in fuzzing systems with embedded protocols is their complex input, which is much more interactive than classical fuzzing targets using a static file format. Nyx-Net addresses this problem by adopting a generative approach, where the user specifies opcodes that are then chained by the fuzzer. Via C code the user is able to implement opcodes to establish connections and opcodes to send packets to the established connections. The input of the sent packets is created by converting packet dumps into a byte format that the fuzzer can effectively process. This approach presents a considerable advantage over grammar-based input generation, as it allows the user to easily capture input packets for the target, in contrast to the process of defining a specific grammar. By combining Nyx-Net's coverage-guided snapshot-based fuzzing technique with its generative input approach, users can easily and efficiently fuzz targets. Two more factors for favoring Nyx-Net is its high usability working with network-based targets and its superior performance compared to similar fuzzers, which is an improving test throughput by up to 300x and coverage by up to 70%. Additionally, Nyx-Net discovered crashes in two of ProFuzzBench's (a benchmark for stateful protocol fuzzing) targets that no other fuzzer had previously detected.

The CAN protocol is chosen as an example to demonstrate the practical application of the developed prototype in this thesis. It is a widely used communication protocol specifically designed for embedded systems and real-time applications. It provides robust and efficient means of communication between electronic control units within a vehicle or industrial setting. With its message-based structure and deterministic behaviour, the CAN protocol enables reliable and high-speed data transmission, making it ideal for applications that require real-time communication and fault tolerance. SocketCAN [16] is a Linux-based networking subsystem that provides a standardized interface for CAN communication. It allows applications to interact with CAN devices by utilising a socket-based API, making it easier to develop CAN-enabled software on Linux platforms. SocketCAN supports various functionalities such as sending and receiving CAN frames, setting filter rules, and managing multiple CAN interfaces, providing a flexible and efficient solution for CAN-based communication. Due to these characteristics, SocketCAN is a suitable choice for demonstrating the functionality of this thesis' prototype.

The following research questions will be answered in this thesis:

- RQ1: What are the key requirements and considerations for adapting the Nyx-Net fuzzer to effectively test and fuzz a program using the protocol implementation SocketCAN, and how can these challenges be addressed?

- RQ2: How can the modifications to Nyx-Net be implemented to preserve its original functionality while incorporating the new changes?
- RQ3: How can the findings and insights gained from successful fuzzing of a target utilising SocketCAN be leveraged to advance fuzzing research on other CAN protocol implementations?
- RQ4: How can the knowledge acquired from successful fuzzing of a target using SocketCAN be applied to advance research on embedded protocols in general?

1.2 Outline

The subsequent sections of this thesis are organized as follows.

Related Work Chapter 2 includes the analysis of related work and existing academic projects to help identify relevant concepts. General fuzzing papers serve as a valuable starting point to gain an overview of the broad field of fuzzing. However, the focus of this thesis is on network and embedded fuzzing, leading to a deeper examination of literature in these specific areas. The selected literature encompasses a range of resources, including survey papers that cover the whole fuzzing domains, as well as proposals and descriptions of specific fuzzing tools.

Background In chapter 3 foundational knowledge about this thesis' prototype fuzzer is presented. Before starting to implement the new prototype, Nyx-Net needs to be set up and executed on the development machine. This involves running Nyx-Net on an Ubuntu 22.04 LTS installation, which may require making necessary adjustments to Nyx-Net's configuration as it differs from the original tested version. To ensure its functionality, Nyx-Net is used to fuzz one of the example targets, dnsMasq, during this stage.

Prior to making any modifications to Nyx-Net, it is imperative to gain a comprehensive understanding of how it operates. A thorough understanding of the steps involved in fuzzing a target program is important. Specifically, areas such as the QEMU setup, network emulation, and the generation of fuzzing input require careful consideration, as these are the aspects that will need modification to enable fuzzing of a target program utilising SocketCAN.

Since this thesis focuses on implementations of the CAN protocol, it is crucial to understand how the target program can be constructed accordingly. In the context of Linux, SocketCAN presents a suitable option, as it offers an implementation of CAN protocols specifically designed for the Linux environment.

Prototype Implementation A detailed explanation of the prototype implementation is described in chapter 4. The general objective is to gradually customize the fuzzer to effectively fuzz a target program that utilises SocketCAN. The process follows an iterative approach, beginning with a straightforward test target program containing a vulnerability that the fuzzer must identify. The target program is then systematically

enhanced, becoming more intricate in each iteration. Throughout this process, the fuzzer is continuously adjusted to accommodate the changes in the target program and ensure its functionality.

A significant portion of the development process is focused on the communication with the target program. Targets, which receive and send message through the SocketCAN interface are dealing with specific types of messages and the configuration of the sockets. To make sure the fuzzer can deal with these distinctions, a more detailed research in the technology needs to be done, followed by necessary adaptations. During fuzzing, the target program operates within a custom version of QEMU. For smoother development, QEMU is launched with enabled CAN support. Among the available options, this thesis employs the setting where QEMU utilises a virtual CAN interface on the host system, requiring manual startup before fuzzing. This setup facilitates monitoring target messages during fuzzing. Once development concludes, CAN support within the virtual machine is no longer essential since Nyx-Net's socket emulation bypasses the CAN interface for data transmission.

Evaluation Chapter 5 includes steps to verify the functionality of the implemented fuzzer in this thesis. Within the scope of this study, this entails verifying that the fuzzer can effectively fuzz a target employing the SocketCAN interface and successfully identify artificially introduced vulnerabilities that lead to program crashes. A range of test targets are supplied, each varying in terms of their functionality and complexity.

Future Work and Research Directions Fuzzing the SocketCAN protocol implementation marks a first step within the broader realm of CAN protocol and embedded protocol fuzzing. Chapter 6 provides the reader with ideas for further expanding the research in the area of embedded protocol fuzzing, especially while using the fuzzer created in this thesis.

CHAPTER 2

Related Work

This chapter provides an overview of the current state of general fuzzing and embedded fuzzing techniques. Section 2.1 presents fundamental concepts in fuzzing, including its application to embedded systems, while also addressing the major challenges of it. Section 2.2 provides an introduction to various fuzzing tools, beginning with the popular and widely adapted AFL fuzzer and its descendants, followed by fuzzers specifically designed for network based targets and embedded systems. When possible, a comparison is drawn with the Nyx-Net fuzzer, which helps to highlight why Nyx-Net serves as the foundation for the prototype developed in this thesis. The chapter ends with section 2.3, which presents various existing fuzzing tools for the CAN protocol.

2.1 Fuzzing Surveys

Fuzzing was first introduced in the early 1990's from Miller et al. [45] and is now one of the most widely used techniques to test software correctness and reliability [44]. In a broad sense, fuzzing involves the iterative execution of a program using generated inputs, which could be potentially malformed either in their syntax or semantics. The primary objective is to discover vulnerabilities within a program. Nowadays, many of the big technology companies include fuzzing in their development practises. Some examples are: Cisco [5], Google [8, 9, 2] and Microsoft [11, 24].

At a high level, fuzzers can be classified into three categories: black-box, grey-box, and white-box [44]. A black-box fuzzer operates without access to the internal workings of the target program. To guide the fuzzing process it relies solely on information obtained from observing the outputs generated by the corresponding inputs. White-box fuzzing creates test cases by examining the internal structure of the target program and leveraging information gathered during its execution. Combining elements of both black- and white-box fuzzers, grey-box fuzzers can access some internal information about the target program. Unlike white-box fuzzers, grey-box fuzzers do not reason about the complete

semantics of the target, instead they might perform lightweight static analysis or gather dynamic information such as code coverage. Grey-box fuzzers rely on approximated and imperfect information to gain speed and effectively test a larger number of inputs.

The three categorise explained above are just one way of differentiating the vast amount of existing fuzzers. There are other aspects, such as:

- Instrumentation: Program instrumentation can be executed either statically or dynamically. Static instrumentation occurs before the target program runs, while dynamic instrumentation takes place while the target program is running.
- Seed Selection: Optimize code coverage by selecting specific seeds.
- Preparing a Driver Application: Sometimes it is more efficient to not fuzz the target program directly, but to prepare a driver for fuzzing. For example IoTFuzzer [26] targets IoT devices by letting the driver communicate with the corresponding smartphone application.
- Scheduling: The objective of scheduling is to analyse the available information on configurations between test cases and choose the one that is more likely to yield favorable outcomes, such as finding a higher number of unique bugs or maximizing code coverage with the generated inputs.
- Input Generation: The technique employed for input generation is a critical design choice in a fuzzer since the content of a test case directly determines whether a bug is triggered or not. There are two types: generation-based and mutation-based. Generation-based fuzzers create test cases based on a specified model that describes the expected inputs for the target program. These are referred to as model-based fuzzers. On the other hand, mutation-based fuzzers generate test cases by modifying existing seed inputs. Such fuzzers are considered model-less because seed inputs are merely examples and do not comprehensively cover the entire input space expected by the target program.

The paper of Zhu et al. [67] describes the knowledge gaps to develop efficient defect detection solutions for fuzzers. The key gaps are briefly summarized as:

- Strict valid input space: As modern applications grow in complexity it gets increasingly harder to generate valid inputs that conform to the requirements of these applications.
- Various targets: The diverse range of targets introduces variations in the execution environment and the types of defects encountered. As a result, effectively fuzzing across this broad spectrum of targets becomes a challenge.

Recent research has been conducted in various aspects of fuzzing. These range from the input generation, such as grammar-based test case generation [39, 61], as well as fuzzing different types of target programs, including kernel [58] and neural network testing [36, 49]. The present thesis concentrates its literature research on fuzzing network based and embedded system targets.

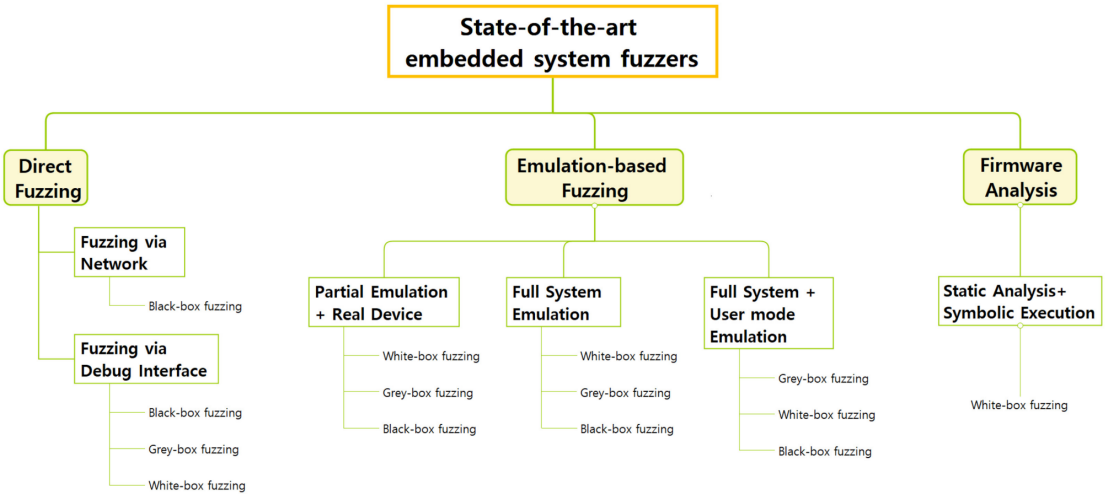

Figure 2.1: Taxonomy of embedded system fuzzers. [62]

Figure 2.1 provides an overview of the taxonomy for embedded system fuzzing, while also including the conventional classification of black-box, grey-box, and white-box fuzzing. In the context of this taxonomy the fuzzer implementation of this thesis can be put into the category of Emulated-based Fuzzing –> Full System Emulation —> Grey-box fuzzing.

In the context of embedded system fuzzers, it is more appropriate to classify them based on their connectivity to the embedded system hosting the target program. Yun et al. [62] emphasized this in their study. As a result, the first types of our taxonomy are direct fuzzing, emulation based fuzzing, and firmware analysis. The direct fuzzing approach involves directly connecting to the target device and testing the system firmware without any intermediary components. This category comprises two types of fuzzing methods. The first type involves fuzzing via a network, which allows testing only network applications and is relatively straightforward, but it limits the fuzzing to a black-box approach. The second type, known as fuzzing via debugging interfaces, such as UART or JTAG, enables testing of the operating system and applications, making it more advantageous. However, this method requires significant effort and might not always be possible due to the removal of debugging interfaces. The emulation based fuzzing approach involves using emulation with firmware extracted from embedded devices. Full emulation of every piece of hardware is not always possible due to restrictions on supported kernels and CPUs. To address this, partial emulation forwards peripheral API requests to the real device. This achieves a similar effect as system-mode emulation, but it requires intensive work to prepare the actual device and forwarding mechanism. Another technique, known as

user-mode emulation, only migrates user-level programs to the host OS using a shared memory state (RAM file) with system-mode emulation. The firmware analysis approach is useful in cases where dynamic analysis or emulation is not feasible. In such scenarios, it is possible to perform static code analysis on the firmware. This approach commonly requires comprehensive program information, thus, only white-box fuzzing is possible.

Recent surveys on embedded system fuzzing [62, 32, 30] outline key differences between traditional and embedded system fuzzing. The main challenges are:

- Strong Hardware Dependency/Diversity: Embedded devices use various microcontrollers and operating systems, whereas desktop systems are typically based on x86-64 CPUs and Linux OS. There are more than 10 common types of microcontrollers used in embedded devices, such as ARM, MIPS and Alpha, and more than 5 different operating systems, including Linux, VxWorks and QNX. Desktop fuzzers work directly on the target system, while embedded system fuzzers work on another system due to the low power and limited resources of embedded devices. To tackle this problem, most embedded system fuzzers use emulation or re-host firmware.
- Crash Detection: Crash detection can be challenging compared to desktop fuzzing. The crash detection ratio in embedded systems is lower, with only 37% compared to desktop systems [46]. Desktop systems benefit from various detection mechanisms like error messages, security warnings, and system logs. In embedded system fuzzing, crash detection is complicated because embedded operating systems often lack fault generation mechanisms, and there might not be output devices like monitors to notice crashes. To address this, embedded fuzzers use alternative methods for crash detection: Liveness checks make it possible to periodically probe the state of the device, memory checking tools can detect security violations, and debug ports can help to gain insights into the device.
- Instrumentation: In embedded systems, obtaining the source code of firmware or applications is often not possible. Moreover, recompilation is not an option due to the diverse CPU architectures and operating systems present in embedded devices. As a result, traditional source code instrumentation becomes challenging in this context. To address this limitation, dynamic binary instrumentation tools like Pin [43] and Valgrind [47] are employed. These tools allow for instrumentation of program binaries during testing. This approach is primarily applicable to general-purpose OS-based devices and may not be suitable for embedded OS-specific devices or devices lacking an operating system.
- Performance and Scalability: Embedded systems face performance and scalability challenges due to their limited computational resources and low power constraints. To address these limitations, several techniques have been developed. Functional upgrades are implemented to enhance performance by changing software while keeping the hardware unchanged. Real-time processing and optimised compilers help overcome computational limitations. Additionally, utilising registers, caches,

and DMA instead of memory can improve performance. Parallelisation is often impossible due to limited resources and economic reasons. As a result, embedded systems fuzzing struggles with scalability, and state-of-the-art solutions do not offer parallel fuzzing support.

2.2 Fuzzing Tools

American fuzzy lop [63] (AFL) is a security-orientated fuzzer that employs compile-time instrumentation and genetic algorithms to automatically discover new test cases that trigger previously unseen states in the target binary. AFL stands out for its practical design, featuring low performance overhead, effective fuzzing strategies, minimal configuration requirements, and seamless handling of complex real-world scenarios like image parsing or file compression libraries. There is a more advanced fork called AFL++ [34] that incorporates numerous enhancements (more speed, better mutation, and better instrumentation) and additional features. However, AFL remains noteworthy due to its significant influence as many fuzzers in recent years have been built upon its foundation. One example is AFLNet [50], a grey-box fuzzer for protocol implementations that utilises mutation techniques, state-feedback, and code-coverage feedback to guide the fuzzing process, and unlike other protocol fuzzers, it relies on a recorded corpus of message exchanges rather than protocol specifications or grammars. AFLNet has some key drawbacks when compare to the fuzzer Nyx-Net. First, using real network connections is significantly slower than the function hijacking approach of Nyx-Net. Secondly, because the fuzzer has no clear point to start a new test case, AFLNet has to wait a manually specified amount of time before a test case is executed. Additionally, when using AFLNet, the user must also develop a cleanup script to ensure that the fuzzing target is in an appropriate state after each test case. These cleanup tasks might involve actions such as rolling back file systems or databases to their original state, ensuring a clean slate for subsequent test cases.

In the past, research had already been conducted to improve the feasibility of fuzzing new target applications. Much of this research centred on generating function-style harnesses and employing snapshots taken at the beginning of a test case. One example of function-style harness research is the work by Babić, et al. [20], which focused on fuzzing C and C++ libraries, particularly the generation of fuzz drivers essential for fuzzing these libraries. Another study addressed the challenge of fuzzing Windows applications, which is still under-represented compared to fuzzing on Unix-like systems. Jung, et al. [41] proposed a practical system called *Winnie* to enhance fuzzing of Windows applications. LibFuzzer [10] is another library for coverage-guided fuzzing, it was replaced by FuzzTest [2], a C++ testing framework for writing and executing fuzz tests. These are property-based tests executed using coverage-guided fuzzing. They are similar to Unit tests but more generic and powerful, for example instead of saying: "for this specific input, that specific output is expected", it is: "for these types of input, this generic property must be true". Examples for snapshot based research are: FirmAfl [66], which extends AFL to fuzz IoT firmware, Dong et al. [28] used snapshots to improve fuzzing Android apps, and

the fuzzer Agamotto [59] used a similar approach to Nyx-Net's incremental snapshots to accelerate kernel-level fuzzing. The big difference between all these mentioned tools and Nyx-Net is that they do not focus on complex network based targets.

Snapfuzz [18] is a fuzzer for network applications. With using a user-mode approach it avoids a lot of complexity Nyx-Net comes with, but with the downside of not being able to fuzz advanced targets like hypervisors.

Aichernig et. al [17] proposed a testing tool for black-box systems, combining automata learning and stateful fuzzing. Automata learning provides enough insight into the black-box system to fuzz targets successfully. Using a grey-box fuzzer allows for more targeted and efficient testing, resulting in a shorter testing cycle and a lower overall cost, thus is preferred in this thesis.

MultiFuzz [64] and SFuzz [27] are fuzzers each focus on specific domains, with MultiFuzz specializing in fuzzing for IoT publish/subscribe protocols, and SFuzz being a slice-based fuzzing tool designed to detect security vulnerabilities in real-time operating systems.

Finding memory corruption vulnerabilities in IoT devices without access of firmware images is done by the tool IoTFuzzer [26]. As many IoT devices are controlled by mobile apps, IoTFuzzer creates fitting fuzzing input with the help of the observed communication between the app and the IoT device. This thesis' prototype should not be restricted to embedded protocols used through a mobile app.

Another fuzzer tackling firmware is Ember-IO from Farrelly et al. [33]. It is an automated testing tool for embedded systems specified on monolithic firmware and supporting firmware built for ARM Cortex-M microcontrollers. The fuzzing tool Fuzzware, a fuzzing tool introduced in a study by Scharnowski et al. [53], specialises in monolithic firmware fuzzing. It is especially efficient in identifying crucial fuzzing inputs by precisely analysing how hardware-generated values are utilised within the firmware. This capability allows Fuzzware to effectively pinpoint the inputs that have the most impact, leading to more efficient and targeted fuzzing results. A recent case study conducted by Scharnowski et al. [54] delves into the domain of fuzzing satellite firmware. This study presents challenges faced in this area, such as the complex boot process commonly found in satellite firmwares, resulting in a slower test case throughput speed. The researchers also configure the firmware fuzzer Fuzzware to work on certain satellite firmware images, which enables them to effectively fuzz and discover bugs.

ZigBee is a widely adopted wireless embedded protocol. Z-Fuzzer [52] is a device-agnostic fuzzing tool for detecting security vulnerabilities in Zigbee implementations. It uses a commercial embedded device simulation and hardware interrupts to interact with the fuzzing tool.

AFLIoT [29] is a on device fuzzing framework for Linux based IoT programs, it uses offset-free binary-level instrumentation to fuzz directly on an IoT device. The on device restriction is not suitable for the aim of this thesis.

Eisele Max et al. [31] utilising the debug interface of microcontrollers to fuzz embedded systems. By setting hardware breakpoints it is possible to extract partial coverage feedback even for uninstrumented binary code. As it does not use virtualisation it is not suitable to use as foundation.

A more comprehensive list of fuzzing tools can be found in the survey papers: [44, 62, 30, 32, 31, 67].

2.3 CAN Fuzzing Research

Lee et al. [42] present a successful attack in which they inject malicious CAN packets into a car's network via Bluetooth. In the first phase of their experiment, they inject fuzzed data exclusively into the data section of a CAN message. For accurate transmission of data with the correct CAN ID, they first listen to normal communication on the CAN bus, and extracting the required CAN IDs. This process is facilitated using freely available internet tools. In the second phase, they employ fuzzing inputs for the entire CAN message. These tests are executed on two commonly used car models, with data injection occurring through the Bluetooth-CAN converter present in the vehicles for maintenance purposes. They observe the behaviour of the attacked vehicles through the instrumentation panels, leaving uncertainty as to whether the displayed behavioural changes actually impacted the cars or merely appeared on the panels.

Nishimura et al. [48] use beStorm [1], a commercial dynamic application security testing tool, to investigate vulnerabilities for the CAN FD protocol. Their implementation is not able to process the response of the component under test after fuzzing input is injected, thus cannot be seen as a full functioning fuzzer. Their test runs are executed with beStorm running on a 64-bit machine having Windows 7 as operating system. This machine is connected through a USB serial cable to the development board where the CAN bus is integrated.

A recent study of Zhang et al. [65] proposes another fuzz testing method for the CAN bus. Their mutation engine is constructed to generate fuzzy messages that are similar to real CAN messages, this ensures that the messages are not rejected by intrusion detection systems. An evaluation on real-world vehicle systems demonstrates the effectiveness of the fuzzer. The fuzzing program consists of two main components, a data generator and an anomaly monitor. It runs on a of-the-shelf computer. The fuzzing target is a real vehicle set into parking mode. To make communication between the fuzzer program and the vehicle possible, a CAN bus controller + transceiver is used.

Fowler et al. [35] introduce a fuzzer to allow for experimentation against a target vehicle's CAN bus, specifically developed for HORIBA MIRA Ltd. and thus not publicly available. The fuzz testing software is crafted on a computer linked to a Vector vehicle simulator, a widely-used equipment in the automotive industry. Utilising a vehicle simulator simplifies the development process as direct access to the target vehicle is unnecessary. To establish

[1] `https://www.beyondsecurity.com/products/bestorm`

a connection between the fuzzer software and the CAN bus, PCAN-USB, a Universal Serial Bus to CAN hardware adaptor is utilised.

The majority of the currently proposed fuzzers for the CAN protocol adopt a similar configuration, as depicted in Figure 2.2. The fuzzer is executed on a machine with a 64-bit desktop operating system. To establish communication between the fuzzer and the test target, which communicates through the CAN protocol, a CAN board is employed. This board encompasses a CAN controller and a CAN transceiver. A USB to CAN adapter, such as the PCAN-USB, is utilised to link the fuzzer's machine to the CAN board. The CAN board is further connected to the CAN bus, thereby interfacing with the test target.

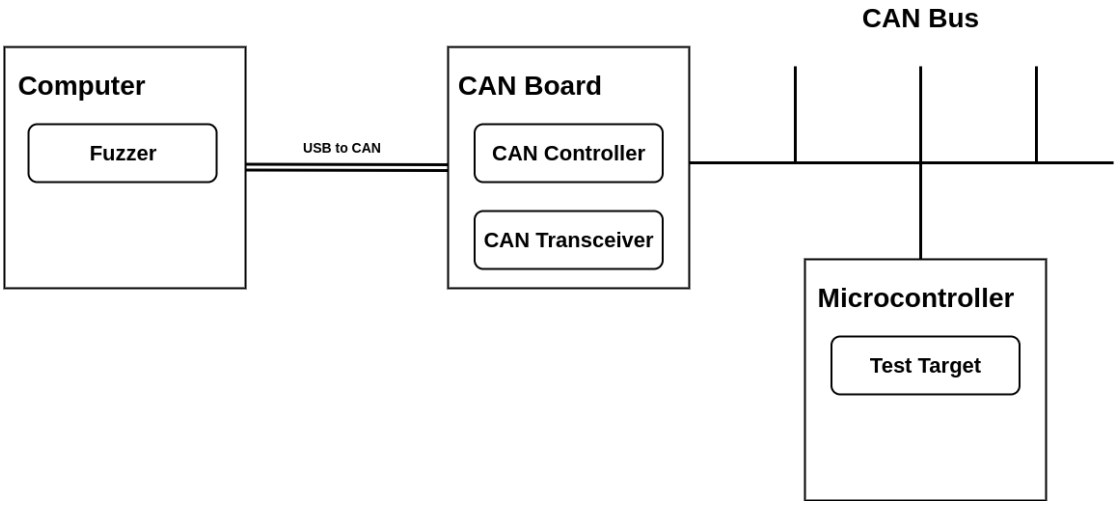

Figure 2.2: Commonly used CAN fuzzing setup.

CHAPTER 3

Background

The following chapter outlines fundamental concepts and theoretical underpinnings that provide a solid foundation for the subsequent chapters of this thesis. The first section 3.1 gives a brief overview of embedded systems. Sections 3.2 to 3.5 describe underlying technologies used in the prototype created within this thesis, such as Controller Area Network (CAN), Virtual Machine (VM) or Quick Emulator (QEMU). Section 3.6 is about Nyx-Net, the foundation of the newly created prototype. It should provide readers with sufficient information to understand the modifications implemented.

3.1 Embedded System

Embedded systems are specialized computer systems designed for specific functions within larger systems or devices [37]. They are typically compact, power-efficient, and operate in real-time or constrained environments. Embedded systems can be found in a wide range of applications, such as consumer electronics, automotive systems, medical devices, and industrial control systems. These systems often have dedicated hardware and firmware tailored to their specific tasks. They are commonly used for controlling and monitoring processes, collecting and processing data, and providing interface functionality. Embedded systems typically consist of a processor, memory, and peripherals. The software responsible for running these systems is stored in read-only memory or flash memory chips [38]. This software, known as firmware, is composed of three main components: the bootloader, an operating system , and a file system [60]. These elements collectively facilitate the proper functioning of the embedded system, enabling it to execute its designated tasks efficiently.

Because they are in some ways accessible by an attacker and are likely to have some kind of vulnerability, the applications running on the embedded system are the most common targets of embedded fuzzers.

Figure 3.1 provides a classification scheme to organize the diverse realm of embedded systems into different categories. This classification helps in understanding and categorising the various types of embedded systems based on specific criteria and characteristics. In their classification Yun et al. [62] are using the criteria of performance, the performance of micro-controllers, and operating system types.

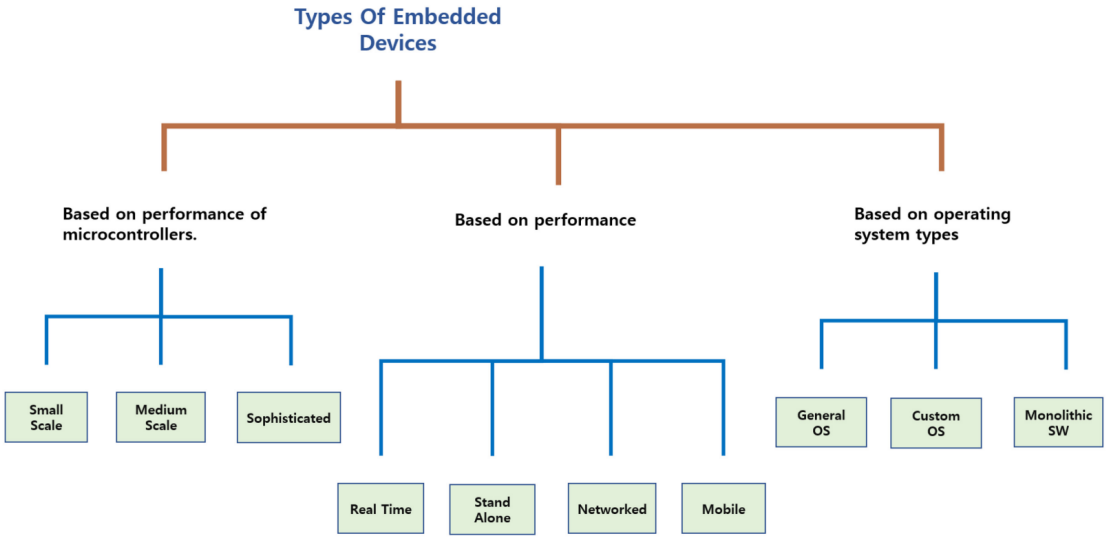

Figure 3.1: Types of embedded devices. [62]

Based on their performance characteristics embedded devices can be divided into four main groups: real-time, stand-alone, networked, and mobile. Additionally, according to the performance of their microcontrollers into three groups: small-scale, medium-scale, and sophisticated embedded devices. The distinction regarding operating system types is done in:

- General-purpose OS: General-purpose operating systems offer a wide range of features, compatibility, and ongoing support from developers, making them suitable for use in embedded devices as well. However, due to the performance constraints of embedded systems, minimal configurations of operating systems are often preferred. For instance, combinations of lightweight user environments like Busybox [1] or uClibc [2], along with a Linux OS kernel, are commonly employed in the embedded domain. This approach allows for optimized performance while still providing essential functionalities required by the embedded devices.
- Custom-built OS: These include real-time operating systems (RTOS) designed for applications that require data processing without buffer delays. These operating systems are ideal for low-power devices. Representative examples of RTOS in

[1] https://busybox.net/
[2] https://www.uclibc.org/

the embedded domain are VxWorks [3] and QNX [4], both widely used in various embedded devices.

- Monolithic Software: Monolithic software serves as both the system and the application. Small-scale devices like SmartCards, GPS receivers, and thermostats often fall into this category. Typically, such devices lack a hardware abstraction layer that hides the physical hardware and provides programming interfaces.

The prototype implementation of this thesis focuses on fuzzing targets which are able to run on a sophisticated microcontroller, is a network target and runs inside a general purpose OS.

3.2 CAN Bus

The CAN bus is a widely used high-integrity serial bus system that enables communication between devices. It is commonly found in automotive and industrial systems. CAN bus operates using a message-based protocol, allowing microcontrollers and devices to transmit and receive data frames.

In modern cars, there can be 70 or more Electronic Control Units that require communication between them. Initially, all components were directly wired, resulting in complex connections. To address this issue, the CAN bus was introduced by Robert Bosch GmbH in 1986. CAN allows individual components to communicate through a centralized bus, simplifying the wiring complexity. Figure 3.2 illustrates the different approaches.

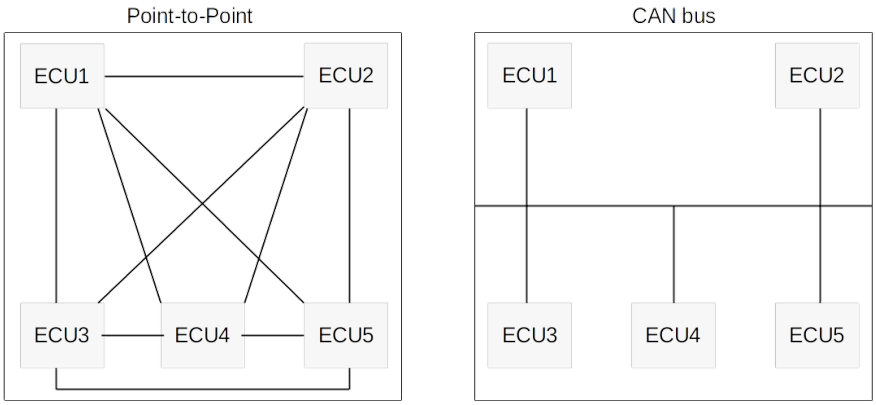

Figure 3.2: Point-to-point communication versus CAN bus approach.

[3] https://www.windriver.com/products/vxworks
[4] https://blackberry.qnx.com/en/products/foundation-software/qnx-rtos

On a CAN bus, data is transmitted in the form of a CAN frame. There are two types, a base frame format with 11 identifier bits and a extended frame format with 29 identifier bits. Figure 3.3 shows a standard frame.

Figure 3.3: Standard CAN frame. [7]

3.3 SocketCAN

The SocketCAN package is an implementation of CAN protocols for Linux. SocketCAN utilises the Berkeley socket API and leverages the Linux network stack to implement CAN device drivers as network interfaces. The CAN socket API is intentionally designed to resemble the TCP/IP protocols, enabling programmers with knowledge of network programming to quickly adapt to using CAN sockets. To communicate over the CAN network a socket [15] has to be created. There are currently two CAN protocols available: the raw socket protocol, and the broadcast manager. After binding [1] (raw protocol) or connecting [6] (broadcast) to the socket, it is possible to read and write from the socket using the commonly known operations. A message sent through the SocketCAN interface has the format of a classical CAN structure as shown in listing 3.1 or the CAN FD structure described in listing 3.2. The classical frame structure is a simplified version of the standard CAN frame seen in figure 3.3. The *can_id* identifies the message sent. The *data* attribute contains the payload and *len* or *can_dlc* representing the length of it. To pass the raw DLC from or to a classical CAN network device the *len8_dlc* element can contain values 9 .. 15 when the real payload length for all DLC values greater or equal to 8. The CAN FD structure is an extension to the classical CAN structure. The main difference is that the data payload can be up to 64 byte in length.

```
1  struct can_frame {
2          canid_t can_id;
3          union {
4                  __u8 len;
5                  __u8 can_dlc; /* deprecated */
6          };
7          __u8    __pad;   /* padding */
8          __u8    __res0;  /* reserved / padding */
9          __u8    len8_dlc; /* optional DLC */
10         __u8    data[8] __attribute__((aligned(8)));
11 };
```

Listing 3.1: CAN frame as specified in the linux kernel (include/linux/can.h).

```
1  struct canfd_frame {
2          canid_t can_id;  /* 32 bit CAN_ID + EFF/RTR/ERR flags */
3          __u8    len;     /* frame payload length in byte */
4          __u8    flags;   /* additional flags for CAN FD */
5          __u8    __res0;  /* reserved / padding */
6          __u8    __res1;  /* reserved / padding */
7          __u8    data[64] __attribute__((aligned(8)));
8  };
```

Listing 3.2: CAN FD frame as specified in the linux kernel (include/linux/can.h).

A virtual CAN interface allows for the transmission and reception of CAN frames without real CAN controller hardware. A virtual SocketCAN interface is used for this thesis' fuzzing approach, as it provides a flexible and convenient way to simulate CAN bus communication.

3.4 Virtualisation

A Virtual Machine (VM) is an emulation of a physical computer, often referred to as a guest, running on a physical machine known as the host.

Virtualisation enables the creation of multiple VMs, each having its own Operating System (OS) and applications, all hosted on a single physical machine. However, VMs cannot directly interact with the physical hardware. Instead, a lightweight software layer called a hypervisor acts as an intermediary, managing communication between the VM and the underlying physical hardware. The hypervisor allocates physical computing resources, such as processors, memory, and storage, to each VM, ensuring that they operate independently and do not interfere with one another.

When utilising a hypervisor on a physical computer or server (also known as a bare metal server), the system can decouple its operating system and applications from the underlying hardware. This enables the division of the system into multiple independent virtual machines.

Each of these virtual machines operates with its own operating system and applications while sharing the original resources of the bare metal server, which are managed by the hypervisor. These resources include memory, RAM, storage, and more. The hypervisor acts as a mediator, effectively allocating the resources of the bare metal server to each virtual machine to ensure they operate without interference.

There are two main types of hypervisors:

- Type 1 hypervisors: Run directly on the physical hardware, replacing the need for an operating system. Specific software is used to create and manage virtual machines on this type of hypervisor. Management tools like VMware's vSphere [5] allow users to select a guest OS for installation in the virtual machine.
- Type 2 hypervisors: Run as applications within a host OS and are commonly used on a single-user desktop. With this type of hypervisor, users manually create virtual machines and then install a guest OS within them. The hypervisor allows users to allocate physical resources to the VMs, manually configuring the number of processor cores and memory they can use.

[5] `https://www.vmware.com/products/vsphere.html`

Figure 3.4 shows an overview of the concept of virtualisation, both with type 1 and type 2 hypervisors.

Typically fuzzers make use of virtualisation to create an isolated and externally controlled environment. In the case of this thesis' prototype, the target programs and a custom component (*Agent*) run inside a virtual machine. As it uses QEMU, it utilises type 2 hypervisor virtualisation.

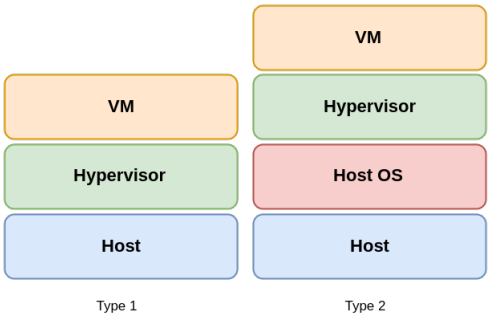

Figure 3.4: Virtualisation with type 1 and type 2 hypervisors.

3.4.1 VMWare Backdoor

The VMWare Backdoor is a feature to increase the performance of virtual machines running on hypervisors. This interface is designed to enhance the communication and coordination between the guest operating system running inside the virtual machine and the VMware hypervisor. This interaction is made faster, through allowing the guest operating system to interact with the hypervisor in a more efficient manner. In the context of this thesis, enabling this feature allows for a more efficient communication between the *Fuzzer* component running on the host and the virtual machine which comprises the *Agent* component and the fuzzing target.

3.4.2 Quick Emulator (QEMU)

QEMU [22] is a versatile machine emulator and virtualiser. It enables the emulation of various hardware platforms, allowing software to run on different architectures and operating systems. The flexibility of QEMU makes it a valuable tool for software development, testing, and debugging, as it allows developers to simulate different environments and configurations. QEMU comes with a CAN bus emulation support [51, 14], which makes it possible to connect individual busses to the host systems CAN API.

The prototype of this thesis, leverages QEMU to create VMs for running target programs. By configuring QEMU to enable CAN support, it becomes possible to run target programs that receive data through the SocketCAN interface. One way to enable CAN support in QEMU ca be seen in listing 3.3.

```
1 -object can-bus,id=canbus0
2 -object can-host-socketcan,id=canhost0,if=can0,canbus=canbus0
3 -device kvaser_pci,canbus=canbus0
```

Listing 3.3: Configuration to enable CAN support in a QEMU VM for qemu-system-x86_64

3.4.3 Kernel-based Virtual Machine (KVM)

KVM is an open-source virtualisation solution for Linux, it functions as a hypervisor and allows running multiple VMs on a single host machine. KVM transforms Linux into a bare-metal hypervisor. Since KVM is integrated into the Linux kernel, it inherently possesses all the necessary components such as memory management, process scheduling, I/O stack, device drivers, security management, and network stack. Each VM created within KVM is treated as a regular Linux process, utilising dedicated virtual hardware resources including network cards, graphics adapters, CPUs, memory, and disks. These VMs are seamlessly scheduled by the standard Linux scheduler, ensuring efficient and reliable operation.

As already done in Nyx-Net, the prototype employs a customized version of QEMU in conjunction with a modified build of KVM to create a VM environment. This optimised configuration guarantees efficient and high-performance virtualisation capabilities.

3.5 LLVM/Clang

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. It simplifies the process of converting source code into machine code. LLVM is able to represent high-level source code using a language-agnostic code called intermediate representation, this enables different programming languages like CUDA or Rust to share analysis and optimization tools before they are translated into machine code. LLVM consists of multiple sub-projects, one of them is Clang. The Clang project offers a language front-end and tooling infrastructure for C family languages, including C, C++, Objective C/C++, OpenCL, CUDA, and RenderScript. It can be used as a replacement for the GNU Compiler Collection, and was started with the intention of better diagnostics, better integration with IDEs, a license that is compatible with commercial products, and a nimble compiler that is easy to develop and maintain. Clang works in three stages: The first stage is responsible for parsing the source code, performing error checks, and generating a language-specific Abstract Syntax Tree (AST). The second

stage optimizes the AST that was generated by stage one. And the third stage, produces the final machine-executable code.

3.6 Nyx-Net Basics

Nyx-Net [57] created by Schumilo, Sergej, et al., is a fuzzer specialising on complex, stateful message-passing systems such as network services. It has demonstrated successful bug discovery in various applications including MySQL Client, Lighttpd, and Firefox. Nyx-Net is built upon two primary design principles. Firstly, hypervisor-based snapshot fuzzing to guarantee fuzzing without interference and fast reset to a clean state. Secondly, selective emulation of network functionality to avoid the creation of slow real network traffic. Their implementation is based on kAFL [56], Redqueen [19] and Nyx [55]. kAFL addresses the challenge of coverage-guided kernel fuzzing by employing an OS-independent and hardware-assisted approach. Using a hypervisor and Intel's Processor Trace technology, it achieves independence from the target operating system, because it only requires a minimal user space component to interact with the target OS. Redqueen is a fast general purpose fuzzer for x86 binary applications. Nyx is a fast coverage-guided hypervisor fuzzer with a mutation engine that utilises custom bytecode programs encoded as directed acyclic graphs and affine types. Nyx's approach provides the possibility to express target input for complex scenarios.

3.6.1 Nyx-Net Components

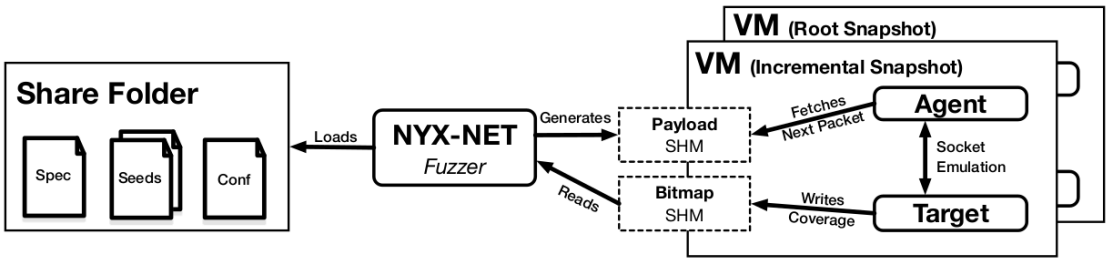

Figure 3.5: Overview of Nyx-Net components. [57]

In figure 3.5 an overview of Nyx-Net's components is shown. This section explains the components involved.

Share folder

The *Share Folder* contains all the essential elements required for fuzzing a particular target, such as the target application, its dependencies, seeds, configurations, and a description of the affine type opcodes that define the fuzzing input. It is located within the *targets* directory.

A bash script may be composed to prepare a target program. This script can contain procedures such as creating a new Linux user or copying files. These scripts should be placed in the *targets/extra_folders/<target>* directory and named *setup.sh*. The building process for a target program can be encapsulated within a bash script as well. Running this script provides the *Fuzzer* with a compiled version of the target program. The tool Nyx-Packer [13] is used for this purpose. It consists of a multitude of tools to ready a specific target program for fuzzing. Users can specify configurations, such as the nyx_net_port, defining the target program's port, the path to the setup scripts mentioned above, and a path directed at the folder containing the fuzzing input creation logic.

The fuzzing input specification of a target is located within the *targets/specs* directory. A file named *nyx_net_spec.py* defines the specification. It includes the payloads data type and its restrictions, a defined kAFL hypercall to create the intermediate snapshots and optional seed inputs the mutation engine can use to create better fuzzing input. During fuzzing the *Agent* component delivers these inputs to all function calls that try to read data from the specified connection, for example *read* or *recv* calls. The directory *targets/specs* includes various examples and in section 4.8 the approach is explained in more detail.

Fuzzer

Before fuzzing starts the *Fuzzer* component loads all relevant information from *Share Folder*. During fuzzing it communicates with the *Agent* and *Target* components which are running inside a custom QEMU VM utilising a modified build of a KVM. The fuzzer maintains a root and incremental snapshot of the same VM. The fuzzing input is stored in a shared memory accessible by the *Agent*, and coverage information from the *Target* is obtained through another shared memory. Running the *Fuzzer* outside of the VM allows it to has full control over the environment.

Nyx-Net utilises a customized version of QEMU, combined with a modified build of Kernel-based Virtual Machine (KVM), to create a Virtual Machine (VM) environment where the fuzzing *Target* and *Agent* run. QEMU is responsible for configuring the VM state and emulating devices, while KVM leverages hardware virtualisation extensions to enable native execution of the guest OS within the VM. This configuration ensures efficient and high-performance virtualisation. The *Fuzzer* component interacts with both QEMU and KVM to gain control over the VM and perform operations such as resetting the state to a specific snapshot.

Agent

Nyx-Net utilises an *Agent* component within the VM to manage the fuzzing cycle. This agent signals the readiness of the *Target* to receive input and decides when to create snapshots. Subsequently, the *Agent* transfers the input provided by the *Fuzzer* to the *Target*, and upon test case completion, notifies the *Fuzzer*. Interactions between *Fuzzer* and *Agent*

are facilitated through hypercalls, which are similar to syscalls but specifically designed for VMs. Socket emulation is done by hooking into existing libc networking functionalities. The code for the *Agent* component is located in *packer/packer/linux_x86_64-userspace*.

The fuzzer Nyx-Net is capable of fuzzing both client and server targets, with the Agent component always taking the opposite role. The socket emulation consists of two main concepts, connections and socket related function hooks. The *Agent* creates a custom connection instance for each socket connection in the target program. This instance includes the client and server socket file descriptors, the server port, and a flag indicating whether the connection is disabled. In cases where the target duplicates file descriptors, up to 8 different file descriptors per client and server can be saved. The number of saved file descriptors for this specific use case is also stored in a separate variable. The type definition of the *Agent's* connection is shown in listing 3.4.

```
1  typedef struct interfaces_s {
2          int server_sockets[8];          // server fds
3          uint8_t server_sockets_num;     // amount server fds
4
5          int client_sockets[8];          // client fds
6          uint8_t client_sockets_num;     // amount client fds
7
8          uint16_t port;                  // server port
9          bool disabled;                  // disabled indication
10 } interfaces_t;
```

Listing 3.4: Typedefinition of a connection in Nyx-Net.

All active connections are saved in a list. The information saved in these connections is updated and accessed throughout the whole fuzzing cycle. It is mainly used from inside the hooks of socket related functions calls, and guides the communication with the target program.

To enhance the speed of network targets during fuzzing and enable the injection of custom fuzzing data, most network APIs are emulated, utilising an LD_PRELOAD interceptor for common libc functions. LD_PRELOAD is a feature in the Linux dynamic linker that allows users to preload shared object files into the address space of a process. The LD_PRELOAD environment variable contains a list of shared object files that are loaded by the dynamic linker before other shared object files. These files, referred to as *preload libraries*, are injected into the address space of a process when it is launched. By specifying LD_PRELOAD, the process's function calls will be redirected to the implementations within the preload libraries instead of the system libraries or other shared object files. This mechanism allows users to override or extend the behaviour of existing functions in a process without modifying its source code.

In Nyx-Net the hijacked functions are located in the directory *packer/packer/linux_x86_64-userspace/src/netfuzz/inject.c*. The scope of the hooks

encompasses a wide range of function calls such as *read* and *recv*, as well as APIs that operate on file descriptors like *dup* and *close*. The hooks closely interact with the connections described in listing 3.4, relying on their information and updating their values when necessary. Nyx-Net emulates the behaviour of the real calls by returning values that mimic their expected outcomes. For instance, in the case of *read*, the number of bytes read is returned on success, while -1 is returned on error.

Target

The *Target* component represents the test program to be fuzzed. It runs together with the agent inside the custom QEMU VM.

3.6.2 Incremental Snapshots

Network applications often maintain state between test cases or have expensive startup routines. Taking a snapshot of the system's state before executing each test case allows for a deterministic reset, independent of the state and complexity of the startup routine. Nyx-Net is able to reset the memory and some limited kernel state of target ring-3 processes to create a snapshot of a whole VM and to reset back to this snapshot after each test. In conventional approaches, creating a snapshot involves duplicating the physical memory and device state. However, due to the potential large size of physical memory, it is desirable to avoid the complete copying and overwriting of memory with every snapshot creation. To address this, Nyx-Net employs a technique where it monitors and tracks the pages within the VM's memory that have been modified since the start of a test case execution. This enables Nyx-Net to avoid overwriting the entire memory during snapshot creation and allows efficient and deterministic fuzzing of a whole OS and even hypervisors. A modern Central Processing Unit (CPU) is equipped with hardware acceleration capabilities that facilitate efficient monitoring of modified pages. The CPU actively tracks and detects when a page is altered. After a certain number of pages have been modified, reaching a predefined threshold (typically up to 512), the CPU interrupts the VM context and informs the hypervisor about the affected pages. The custom KVM used by Nyx-Net, saves these modified pages in a stack, allowing the fuzzer to avoid searching for pages which need to be reset. Furthermore, Nyx-Net incorporates a specialized reset mechanism for emulated devices, which offers significantly improved performance compared to the default device serialisation and deserialisation routine of QEMU.

When fuzzing complex target programs, it is common to encounter specific program states that require a lengthy sequence of received messages to reach. For instance, if a stream of 120 packets needs to be fuzzed, it may be more efficient to focus on mutating and executing the input for only the last 20 packets in each test case. With Nyx-Net this optimisation can be achieved by utilising incremental snapshots, allowing for the creation and removal of snapshots after processing specific portions of the input.

3.6.3 Fuzzing Example Server

The provided pseudocode in Listing 3.5 demonstrates the implementation of a target program that emulates traditional server logic. The subsequent portion of this section outlines the interaction between the fuzzer and the target program. This approach is also applicable to other targets, making this description a guideline for understanding the inner workings of the fuzzing process within the *Agent* component.

```
1 // ... create and bind socket to port 8080 ...
2 listen(server_fd, 3)
3
4 new_socket = accept(server_fd, (struct sockaddr*)&address,
5                      (socklen_t*)&addrlen)
6
7 valread = read(new_socket, buffer, 1024);
8
9 // ... do something with received data ...
```

Listing 3.5: Pseudo server example.

The first injection of custom logic occurs when the target program invokes the listen function on line 2. It calls the real *listen* [6] function and in the end forwards it's return value to the target program. Additionally, if not previously done, creates a new connection (as defined in listing 4.9) with the port the socket file descriptor *server_fd* is bind to. If another connection already exists for this port, it is updated rather than created anew, ensuring that each port associated with the target program has only one corresponding connection. Subsequent updates related to the connection with port 8080 will be managed by this existing connection. The hook on the *listen* function also includes code to create a new socket and connects it to the server address. As the *Agent* plays the role of a client in this scenario, it also adds this newly created socket to the *client_sockets* attribute of the connection.

During the hook of the call accept [7], the socket *server_fd* is added to the *server_sockets* attribute of the connection.

At the time of reading, the *Agent* has one active connection with *server_sockets* containing the socket file descriptor created by the target program, *server_sockets_num* and *client_sockets_num* set to one, *client_sockets* containing the socket file descriptor created by the *Agent* component itself, the *port* set to 8080 and both *is_can* and *disabled* set to false.

While hooking the *read* [8] call, the *Agent* directly writes the data generated by the mutation engine to the memory location pointed to by the variable *buffer*. This data is

[6] https://man7.org/linux/man-pages/man2/listen.2.html
[7] https://man7.org/linux/man-pages/man2/accept.2.html
[8] https://man7.org/linux/man-pages/man2/read.2.html

subsequently available for further processing within the target program.

CHAPTER 4

Prototype Implementation

The fuzzer Nyx-Net is constructed with the help of the Nyx framework [1]. This framework consists of various modules which can be used to build a custom fuzzer. In this thesis' prototype the following three modules are used:

- **Spec-Fuzzer** [2]: Serves as a fuzzing frontend with an advantage in handling complex interactive sequences of actions, such as network packets, hypercalls, syscalls, api-calls, or GUI interactions. This is achieved by expressing these actions in a specification, which defines a set of functions with affine types, similar to Rust functions. By chaining and combining these functions, the fuzzer generates sequences of validly typed function calls. The specification is compiled into a straightforward header-only bytecode interpreter. The format of this fuzzer allows for the utilisation of incremental snapshots, which significantly accelerates the fuzzing process.

- **QEMU-Nyx** [3]: is a modified version of QEMU. It consists of several enhancements to QEMU to enable various functionalities, such as Hypervisor-based snapshots, Intel-PT based tracing, and REDQUEEN style magic byte resolution. The extensions make it possible to reset memory and devices, thus providing precise disassembly of the running code even when some parts are swapped out or unavailable. Additionally, it facilitates Intel-PT decoding and allows code instrumentation with breakpoint-based hooks. Moreover, it establishes seamless communication with a fuzzing frontend, which can be based on libnyx [4], enabling a more comprehensive and efficient fuzzing process.

[1] https://github.com/nyx-fuzz/nyx
[2] https://github.com/nyx-fuzz/spec-fuzzer
[3] https://github.com/nyx-fuzz/QEMU-Nyx
[4] https://github.com/nyx-fuzz/libnyx

- **Nyx-Packer** [5]: Is a packing tool designed specifically for Nyx virtual machines, including those used in Nyx-Net. It serves as a versatile utility, capable of executing a range of tasks to create a fully functional and comprehensive *Share Folder*. This *Share Folder* houses all the data required to initiate and efficiently run the virtual machines during the fuzzing process.

For the modules *Spec-Fuzzer* and *Nyx-Packer* it was not possible to update without main changes, thus the same version where used as in the publication of the fuzzer Nyx-Net. The module *QEMU-Nyx* is updated to the version of 13. April 2023.

This chapter describes the implementation of the prototype created within this thesis, it aims to fuzz target programs utilising the SocketCAN interface. The chapter starts with providing the reader with an overview of the introduced changes in section 4.1. Then, the development setup is described in section 4.2. Sections 4.3 and 4.4 outline the changes made regarding the VM setup. Sections 4.5 and 4.6 introduce new configuration options. In section 4.7 to 4.9 details about the fuzzing process are specified and section 4.10 shows how a crash found by the fuzzer can be reproduced manually. Section 4.11 ends this chapter with a description of fuzzing an example target program which uses SocketCAN.

4.1 Changes Overview

To gain insight into the alterations made to Nyx-Net and to grasp the details discussed in subsequent sections, figure 4.1 presents the newly introduced changes. It incorporates the Nyx-Net overview depicted in figure 3.5 and highlights the additions and modifications. The adaptions are distinguishable with a distinct font style and are highlighted in red. Whenever an entire component is altered or a new one is introduced, it is encompassed within a blue box for clear identification.

To enhance the convenience of working with the fuzzer following user scripts are added:

- **all_canlinux.sh**: Takes care of compiling all the necessary components and initiating the fuzzing process of a SocketCAN target.
- **canreceive_rust_fuzzer_debug.sh**: After a crash is discovered, this file can be executed to generate a more readable crash input.
- **can-send-data.py**: Proves useful in sending the crash input to a target program running on the host, allowing for easy reproduction and further investigation of the crash.
- **all_dnsmasq.sh**: Compile and fuzz the target dnsMasq, which was already part of Nyx-Net.

[5] `https://github.com/nyx-fuzz/packer`

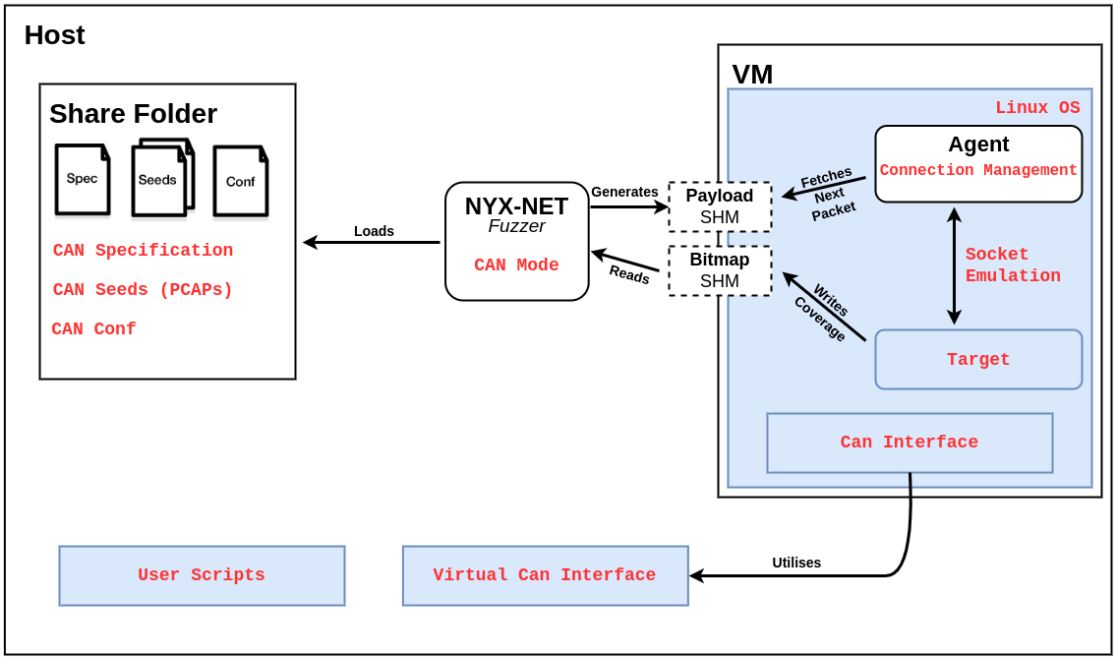

Figure 4.1: Overview of changes within the new fuzzer. Changes are marked red and blue.

Inside the *Share Folder* the specifications, seeds and configurations have changed. A new specification for fuzzing input has been introduced, specifically designed for handling the creation of data sent over a CAN interface. This updated specification makes use of multiple seed inputs in the form of PCAP files, each containing relevant information about CAN messages suitable for target programs that utilise a CAN interface. Additionally, certain modifications have been made to the configuration process when fuzzing CAN targets. These changes to the specifications, seeds, and configurations have been implemented to enhance the efficacy and success of the fuzzing process, when dealing with CAN targets.

The new fuzzer provides the option for users to enable a CAN mode when necessary. It is essential to enable this mode only when the target program utilises a CAN interface, as otherwise it could lead to undesired behaviour especially in regards to the communication from the *Agent* to the *Target*. Inappropriately injecting fuzzing input of type *can_frame* or *canfd_frame* when the communication does not occur through a CAN interface is not desirable. Therefore, it is important to ensure that this mode is activated only when the CAN interface is in use by the target program. By default, the newly introduced CAN mode remains disabled, ensuring that the fuzzer operates just as it did before all these modifications were made.

The virtual machine's operating system has undergone a significant overhaul, resulting in a fresh, minimalistic Linux installation with integrated CAN bus support. Additionally,

the Linux Kernel has been upgraded to a newer version. These changes now enable the option to start the operating system with or without CAN support, ensuring that the original functionality of Nyx-Net remains intact without any disruption. When CAN support is enabled, a CAN interface is active within the operating system, allowing target programs to interact with it. This CAN interface within the virtual machine communicates with a corresponding virtual CAN interface on the host system.

The *Agent* component has undergone most of its updates regarding the connection management and socket emulation. These changes are active only if the CAN mode is enabled. Regarding the connections to the target, the logic for creating new ones and determining the existence of a them has been altered. As for the socket emulation, the primary focus has been on providing appropriate fuzzing input to the target program. Although the *Agent* still utilises system call hooks to inject input, the process is not as straightforward as before. While there have been considerable changes, it's essential to note that not the entire *Agent's* code has been overhauled, as much of the existing logic from Nyx-Net remains applicable to targets utilising a CAN interface.

4.2 Local Setup

Before making any adaptations to the fuzzer Nyx-Net, it is essential to verify that it is set up to carry out the fuzzing process successfully. This is ensured by enabling Nyx-Net to fuzz one of the example targets provided. The development environment comprises a machine with Ubuntu 22.04 LTS and Linux kernel version 5.19.0-46-generic. This section outlines all the necessary changes to enable successful fuzzing with Nyx-Net on that machine.

To begin the setup process, it is recommended to start with a clean installation of Ubuntu. The packages that need to be installed include *pkg-config, build-essential, curl* and *libgtk-3-dev*, without these packages the fuzzer will not compile.

Compilation errors

There is a problem when using a Rust version greater than 1.53.0 for compiling the fuzzer. An error is thrown that states *reference to packed field is unaligned*. A more comprehensive explanation of this issue can be found in the tracking issue on GitHub [6]. As a temporary solution to this problem, it is currently recommended to install and use Rust version 1.53.0. This can be achieved by executing the commands provided in listing 4.1.

```
1 curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
2 sudo snap install rustup --classic
3 rustup install 1.53.0
4 rustup override set 1.53.0
```

Listing 4.1: Commands to install Rust version 1.53.0.

Another encountered error while running the fuzzer is related to the usage of the *transmute* function in Rust. In Rust, the *transmute* function allows for reinterpretation of data types, but is restricted to constant expressions and static variables to ensure safety. The specific error message displayed is: *'transmute' is only allowed in constants and statics for now*. The issue lies in the file *.cargo/registry/src/github.com-1ecc6299db9ec823/nix-0.26.2/src/sys/time.rs*. The *.cargo* folder is automatically generated by Cargo, the Rust package manager, and contains essential configuration files and project-specific data. To resolve this problem, the nix package, where the error occurs, needs to be downgraded to version 0.25.1 or lower. This is necessary because Rust version 1.53.0 is required, and it is not compatible with the latest version of nix (0.26.2).

[6] https://github.com/rust-lang/rust/issues/82523

VMWare backdoor

To ensure proper functioning of the fuzzer, it is necessary to enable the VMware backdoor. It is a feature to increase the performance of virtual machines running on hypervisors within the KVM kernel module. (see section 3.4.1 for a detailed description)

Activating the VMware backdoor involves passing specific parameters to the KVM modules during the reloading process. Enabling this virtualisation support can be accomplished by using the commands provided in listing 4.2. The modules *kvm* and *kvm-intel* are getting removed, and then added again with the option *enable_vmware_backdoor* set to yes. By following these steps, the fuzzer can fully harness the capabilities of the VMware backdoor, optimizing the performance of virtual machines and enhancing the overall efficiency of the fuzzing process.

```
1 sudo modprobe -r kvm-intel
2 sudo modprobe -r kvm
3 sudo modprobe  kvm enable_vmware_backdoor=y
4 sudo modprobe  kvm-intel
```

Listing 4.2: Commands to enable VMware backdoor support.

LLVM version

LLVM is a powerful compiler infrastructure designed to optimize and compile source code for various CPUs. (see section 3.5 for a detailed description) Ubuntu 22.04 comes pre-installed with LLVM version 14. However, since Nyx-Net utilises AFL, which requires LLVM with a version lower than 12, it becomes necessary to manually install the compatible LLVM version to ensure the successful compilation of AFL, hence the correct fuzzing with the fuzzer. In the development setup, LLVM version 11 is installed.

Listing 4.3 provides a comprehensive and step-by-step procedure to set up LLVM and Clang with version 11. The first step involves configuring the system to be able to fetch and install packages from the LLVM repository. To be able to install LLVM packages especially with version 11, the repository *llvm-toolchain-bionic-11* is added to the list of software sources used by the package manager in Ubuntu. In the next step, the individual packages are installed, which are *llvm-11, lldb-11, llvm-11-de* and *libllvm11*. After successfully installing the required LLVM packages, the *update-alternatives* command is used to update the alternatives for the *llvm-config* command in Ubuntu. This allows users to switch between different versions of *llvm-config* in a convenient way. In this example the version 11 of the command is added as alternative, and also set as default.

A similar process is followed to set up Clang with version 11. Clang is a part of the LLVM project and serves as a language front-end and tooling infrastructure for languages in the C family. First the needed packages (*clang-11, lldb-11, lld-11*) are installed, and with the commands *update-alternatives* and *ln*, the default is set to version 11 whenever Clang is used.

```
1  sudo wget --no-check-certificate -O \
2    - https://apt.llvm.org/llvm-snapshot.gpg.key \
3    | sudo apt-key add - && \
4  sudo add-apt-repository 'deb␣http://apt.llvm.org/bionic/␣\
5  ␣␣llvm-toolchain-bionic-11␣␣main' && \
6  sudo apt-get install llvm-11 lldb-11 llvm-11-dev libllvm11 \
7    llvm-11-runtime -y && \
8  sudo update-alternatives --install /usr/bin/llvm-config \
9    llvm-config /usr/bin/llvm-config-11 11 && \
10 sudo update-alternatives --config llvm-config && \
11 sudo apt install clang-11 lldb-11 lld-11 -y && \
12 sudo update-alternatives --install /usr/bin/cc cc \
13   /usr/bin/clang-11 100 && \
14 sudo update-alternatives --install /usr/bin/c++ c++ \
15   /usr/bin/clang++-11 100 && \
16 sudo ln -s /usr/bin/clang-11 /usr/bin/clang && \
17 sudo ln -s /usr/bin/clang++-11 /usr/bin/clang++
```

Listing 4.3: Commands to install LLVM and Clang version 11.

4.3 QEMU CAN Support

To ease the development of the fuzzer on a target utilising SocketCAN, CAN support within the QEMU environment is enabled. This configuration allows programs running inside the VM to access and interact with a running CAN interface on the host system. When initialising a QEMU VM, the *Fuzzer* component configures the virtual environment. It must start the VMs with the correct parameters which are shown in the code snippet 4.4.

```
1 cmd.push("-object".to_string());
2 cmd.push("can-bus,id=canbus0".to_string());
3 cmd.push("-object".to_string());
4 cmd.push("can-host-socketcan,id=canhost0,if=can0,
5 ␣␣␣␣␣␣␣␣␣␣canbus=canbus0".to_string());
6 cmd.push("-device".to_string());
7 cmd.push("kvaser_pci,canbus=canbus0".to_string());
```

Listing 4.4: Changes in the fuzzer component (fuzzer/libnyx/-fuzz_runner/src/nyx/params.rs) to enable CAN support in all VMs.

In order for programs within the VM to utilise the CAN interface of the host system, two additional steps must be executed. First, a CAN interface needs to be set up with a defined bitrate, the bitrate determines the speed of data transmission on the CAN bus, indicating the number of bits that can be transmitted per second. The second step involves bringing up the CAN interface, activating it for data communication. Both of these steps must be performed within the running VM. To accomplish this, the code provided in listing 4.5 is added to the fuzzing setup script located in *targets/extra_folders/canlinux/setup.sh*. The used *ip link* command is a Linux networking utility used for configuring and managing network interfaces on a system. It allows users to view, add, remove, and modify network interfaces, as well as set their properties and states.

```
1 // set bitrate
2 ip link set can0 type can bitrate 1000000
3 // init CAN interface
4 ip link set can0 up
```

Listing 4.5: Setup for enabling CAN inside the VM.

The easiest option to listen to messages send and received from a target program is the command line tool *candump* [7]. It is part of *can-utils* [8], a collection of useful tools regarding SocketCAN. A more detailed description of available tools can be found in section 5.5.1.

[7] https://manpages.debian.org/testing/can-utils/candump.1.en.html
[8] https://github.com/linux-can/can-utils

4.4 Minimal Linux

The *Agent* and *Target* components are executed within a VM that runs a minimal Linux installation supporting the CAN interface. This minimal Linux uses a Linux kernel version 6.2.11 with CAN bus subsystem support. This system was created with the help of Buildroot [9], a tool designed for generating embedded Linux systems through cross-compilation. Relevant configuration details can be found in listing 4.6, it uses the sample toolchain *x86_64-unknown-linux-gnu* from the toolchain generator crosstool-NG [10] and enables recent kernel headers and glibc support. Additionally, Buildroot is instructed to generate a cpio archive for the root filesystem and enables the *iproute2* [11] utilities.

```
1  Toolchain -> Toolchain type -> External toolchain
2  Toolchain -> Toolchain -> Custom toolchain
3  Toolchain -> Toolchain origin -> Pre-installed toolchain
4  Toolchain -> Toolchain path -> ~/x-tools/x86_64-unknown-linux-gnu
5  Toolchain -> Toolchain prefix -> x86_64-unknown-linux-gnu
6  Toolchain -> External toolchain kernel headers series
7            -> 6.1.x or later
8  Toolchain -> External toolchain C library -> glibc
9  Filesystem images -> cpio the root filesystem -> yes
10 Filesystem images -> cpio the root filesystem
11                   -> Compression method -> gzip
12 Target packages -> Networking applications -> iproute2 -> yes
```

Listing 4.6: Minimal Linux configuration used by the new prototype fuzzer.

4.5 CAN_MODE

In order to avoid disrupting Nyx-Net's functionality while introducing new features, a new environment variable called CAN_MODE is introduced. If this variable is enabled, the fuzzer utilises different logic during the fuzzing process. To activate the CAN_MODE the user must provide the option *nyx_net_can* when using the Nyx-Packer for creating the *Share Folder*.

By employing this mode, the fuzzer is also capable of fuzzing targets that simultaneously utilise traditional sockets (e.g., with domain AF_INET) and sockets employing a CAN interface. In this scenario, the fuzzer primarily generates appropriate inputs for the CAN sockets, but also injects data into the other type of sockets. This functionality can be modified by adjusting the specification of the target input (see section 4.8 for more information). Given the focus of this thesis on SocketCAN targets, this approach appears to be fitting.

[9] https://buildroot.org/
[10] https://crosstool-ng.github.io/
[11] https://wiki.linuxfoundation.org/networking/iproute2

4.6 CAN_SPEC_TEST_MODE

To assess the efficacy of the formulated input specification for SocketCAN targets, a new mode named CAN_SPEC_TEST_MODE is introduced. This mode functions as an environment variable, analog to the behaviour of CAN_MODE, and can be configured using the *can_spec_test_mode* option within the Nyx-Packer tool when generating the designated *Share Folder*. This enhancement adds a practical dimension to the testing process, allowing for user friendly assessments of the input specification within SocketCAN targets.

When this environment variable is set, the fuzzer does not hook the libc syscall write, and thus it is possible for a target program to write data on the CAN interface inside the virtual machine. From the interface on the host system, the messages can then be observed and analyzed.

By executing the file *all_can_spec_test.sh*, users have the option to activate this environment variable and subsequently generate an output of all data produced through the input specification. This file serves as a trigger for a series of tasks, encompassing:

- Initiate the fuzzing process for the target program *can_spec_test.c*. This particular file is designed solely to utilise the system call write [12] for sending the received CAN messages through the CAN interface.
- On the host system a new terminal is opened and the command *candump any* is executed. This command will generate output showcasing all the messages transmitted by the target program running within the virtual machine of the fuzzer, and thus all the fuzzing data created with the help of the defined target input specification.
- By monitoring the terminal where *candump* is running, all the generated input which is being injected into the target program can be observed. If necessary, it is possible to modify the *candump* command to record the observed messages into a file, facilitating subsequent analysis.

[12] `https://man7.org/linux/man-pages/man2/write.2.html`

4.7 Fuzzing Workflow

The fuzzing workflow for a target is depicted in figure 4.2. Detailed explanations of each step are presented in the related subsections, which will additionally guide the reader on making any necessary adaptations. To facilitate the execution of all the required steps, it is enough to run the file *all_canlinux.sh*.

Figure 4.2: Fuzzing workflow.

Setup

Running *setup.sh* compiles an AFL clang compiler, the custom version of QEMU and two additional tools it depends on: capstone [13], a disassembly framework, and libxdc [14], an Intel-PT decoding library. These tools are used to analyse and understand the behaviour of the target programs during the fuzzing procedure. It also compiles the *Fuzzer* component, which incorporates an additional debugging version specially designed for reproducing any crashes encountered during the fuzzing process. The procedure for reproducing such crashes is thoroughly explained in section 4.10. Furthermore, running *setup.sh* involves the process of preparing the initial RAM file system for starting the VMs.

Build target

Executing the file *targets/setup_scripts/canlinux/build.sh* uses the AFL clang compiler to build the target program.

[13] `https://github.com/capstone-engine/capstone`
[14] `https://github.com/nyx-fuzz/libxdc`

Create share folder

Running file *targets/packer_scripts/pack_canlinux.sh* uses the Nyx-Packer tool to create the *Share Folder* for the target program. The invocation to the packer is shown in listing 4.7. The option *–nyx_net_can* enables the CAN_MODE, *-spec* takes a path pointing to the specification of suitable fuzzing input, and the path in *–setup_folder* contains necessary files to prepare the fuzzer. Once the process is complete, the *Share Folder* is prepared and ready for utilisation by the *Fuzzer* component.

```
1 python3 nyx_packer.py <path to compiled target> <output path> m64 \
2 —afl_mode \
3 —purge \
4 —nyx_net \
5 —nyx_net_can \
6 –spec ../specs/canlinux/ \
7 —setup_folder ../extra_folders/canlinux && \
```

Listing 4.7: Packacking of a SocketCAN target program using the Nyx-Packer tool.

The last step before running the fuzzer involves generating the configuration for the VM and the Kernel. This is accomplished by running the file *nyx_config_gen.py*.

Fuzz

In the directory *fuzzer/rust_fuzzer* run the command
cargo run –release –verbose – -s ../../targets/packed_targets/nyx_canlinux/.

4.8 Target Program Input Specification

The input specification for a target program that receives data through a SocketCAN interface can be found in the directory *targets/specs/canlinux*. It includes:

- seed inputs
- the file *send_code_raw.include.c*, specifying how the *Agent* receives the fuzzing input
- the file *nyx_net_spec.py*, containing the specification logic

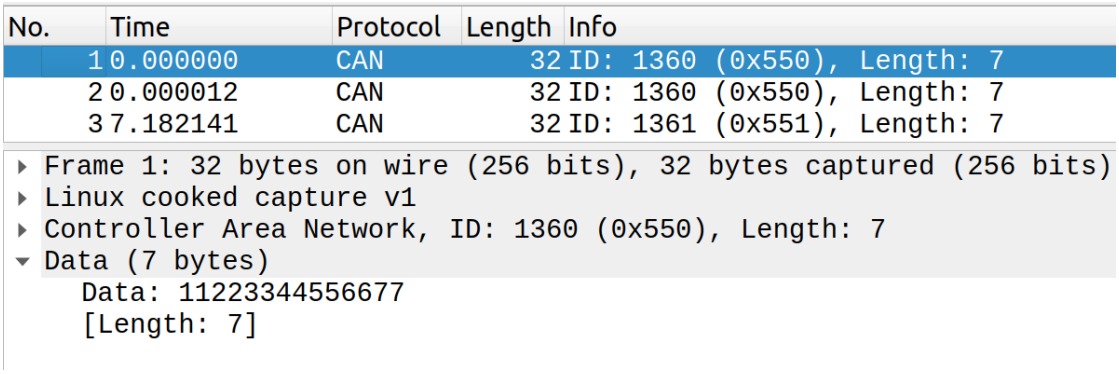

Figure 4.3: Wireshark trace of CAN messages.

The seed inputs consist of PCAP files containing sets of CAN messages. Figure 4.3 displays an example output in Wireshark before being exported as PCAP files.

Executing the file *send_code_raw.include.c* copies the fuzzing input values created by the mutation engine to a memory location the *Agent* component is able to access.

The specification of the fuzzing input is defined in the Python file *nyx_net_spec.py*. This file serves as the blueprint for generating target inputs used during the fuzzing process. The specification is designed as a directed acyclic graph, consisting of multiple interconnected nodes that define the structure and rules for creating valid inputs. When dealing with SocketCAN targets, the specification involves two main nodes. The first node is responsible for creating snapshots, utilising kAFL hypercalls, which provides developers with a convenient way to manage and manipulate snapshots during the fuzzing process. These snapshots are essential for preserving the target program's state and ensuring reproducibility during fuzzing. The other node contains the input definition, including its data type, data restrictions, and the content of the file *send_code_raw.include.c*. This node ensures that the generated inputs adhere to the expected format and constraints.

The payload of the seed inputs provided as PCAP files can be extracted using a few lines of Python code, as demonstrated in listing 4.8. This capability allows for easy integration of existing data and real-world scenarios into the fuzzing process. The resulting specification is then made accessible within the *Share Folder*, ready to be utilised by the *Fuzzer* component.

```
1  for path in glob.glob("pcaps/*.pcap"):
2      cap = pyshark.FileCapture(
3                          path, display_filter="can",
4                          include_raw=True, use_json=True
5              )
6
7      for pkt in cap:
8          if int(pkt.data.len) != 0:
9              data = bytearray.fromhex(pkt.data.data_raw[0])
10             # ... add data to mutation engine ...
11     cap.close()
```

Listing 4.8: Example of extracting payload data from CAN messages exported as PCAP files.

Add new target

For simple SocketCAN target programs it should be enough to put the source code and everything needed to build the program into the folder *additional-files/canLinuxExample*. For more advanced target's it is possible to add and adapt certain configurations. This document should provide the reader with enough information for doing that, especially the specific steps highlighted in section 4.7 are directing the reader to the most critical aspects. The high level process of fuzzing a new advanced target program is to run the program locally, transmit packets using tools such as cansend [4] or similar utilities, trace the sent packets using tools like Wireshark, and transform the traced packets into suitable input the fuzzing engine can work with.

4.9 Agent

Figure 4.4 presents a high-level overview of the *Agent's* socket emulation, offering clarity on how communication with the *Target* component is done. This diagram aims to simplify the understanding of the more detailed explanations regarding the functioning of the *Agent* component.

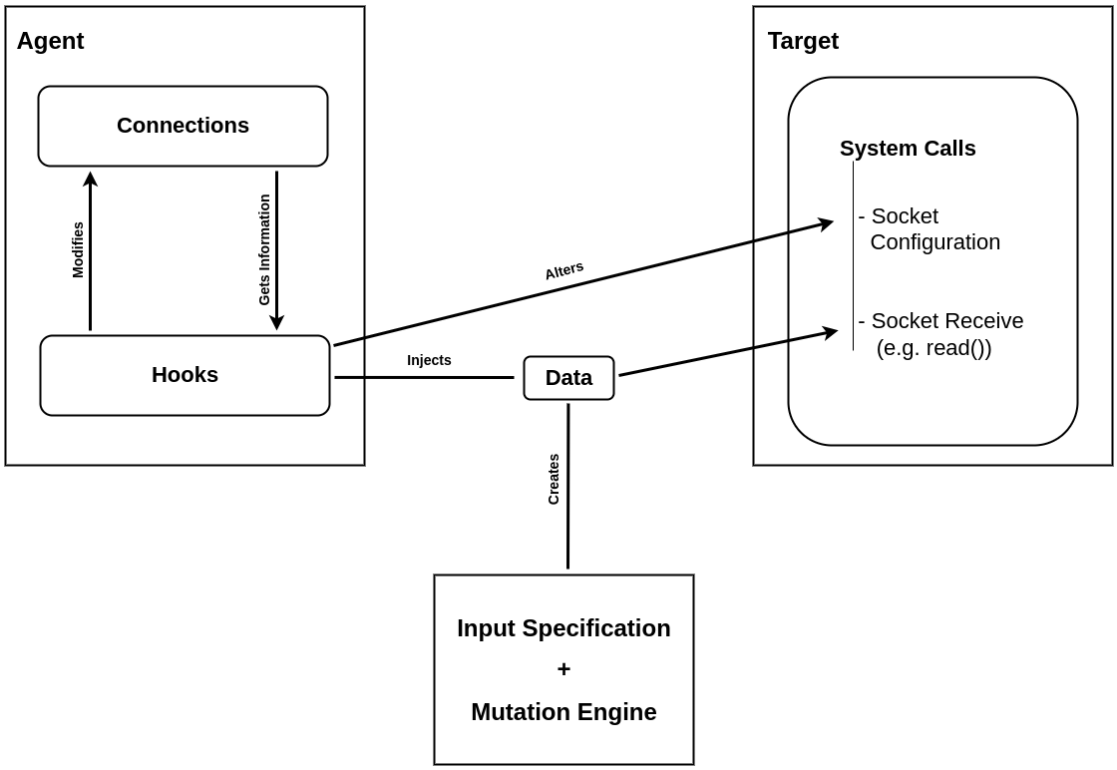

Figure 4.4: Agent socket emulation overview inside the fuzzer.

The *Agent* component maintains a list of all active connections established with the target. These connections store essential information about the associated socket and the nature of the connection. The *Agent* effectively utilises libc function hooks to modify connection attributes, but also to receive certain information.

The libc functions within the target program can be classified into two domains. Firstly, there are functions responsible for creating, updating, or closing sockets, primarily hooked by the *Agent* to prepare for injecting the appropriate fuzzing input. Secondly, there are libc function calls used by the target program to receive data, such as the read [15] function. By intercepting these calls, the hooks are changing the functions behaviour with the end goal of injecting data into the target. The data is provided by the mutation engine, which is using the input specification to create suitable fuzzing input.

[15] `https://man7.org/linux/man-pages/man2/read.2.html`

4.9.1 Connection Management

As described in section 3.6.1 the *Agent* holds a list of custom connections (see 3.4) including important information for guiding the fuzzing process. However, when dealing with SocketCAN target programs, specific adaptations need to be made to those connections. Unlike regular target programs, which are bound to specific ports, SocketCAN target programs receive data through the CAN interface, making port-based identification unsuitable for the program logic. To overcome this challenge, new functionality for the connections is introduced. A boolean variable is now included in the connection structure, serving as an indicator of a CAN connection. In *CAN_MODE*, the fuzzer relies on this new boolean variable to determine the presence of a connection instead of the port, which was the norm before these changes were introduced. The updated definition of the adapted connection structure can be observed in listing 4.9. These adaptations ensure that the fuzzer remains flexible and capable of handling different types of target programs, including those utilising the SocketCAN interface.

```
1 typedef struct interfaces_s {
2         int server_sockets[8];       // server socket fds
3         uint8_t server_sockets_num; // amount server sockets
4         int client_sockets[8];       // client socket fds
5         uint8_t client_sockets_num; // amount client sockets
6         uint16_t port;
7         bool is_can;
8         bool disabled;
9 } interfaces_t;
```

Listing 4.9: Adapted type definition representing a connection within the new fuzzer.

4.9.2 Target Input Injection

```
1 read(int fd, void *buf, size_t count)
```

Listing 4.10: Read syscall definition.

In scenarios where the target program does not receive data via the SocketCAN interface, the fuzzer's *Agent* component is equipped to directly write the payload into the buffer utilised by the libc calls. For instance, in case of the *read* function, as depicted in listing 4.10, the fuzzer would typically inject the fuzzing input directly into the memory address pointed to by *buf* parameter. However, when the target program utilises the SocketCAN interface, the structure pointed to by the *buf* parameter takes the form of a *can_frame* 3.1 or a *canfd_frame* 3.2. The problem is now that the fuzzers mutation engine generates values solely for the *data* attribute of these two structs. To tackle this issue the *Agent* constructs a CAN(FD) specific frame and writes it to the address of *buf* when running in CAN_MODE. Doing it only while CAN_MODE is enabled the adaptions do not interfere with the logic of fuzzing classical socket based targets. In the current implementation the CAN(FD)'s ID, length and data attributes are set. This is enough to fuzz most of the SocketCAN targets. To get the ID needed for constructing the frame, the *Agent* hooks the system call getsockopt [16]. This allows the *Agent* to retrieve a variable of type *can_filter*, which holds the CAN filters defined by the target program. A sample code for retrieving the ID is shown in listing 4.11.

```
1 struct can_filter filter;
2 socklen_t optlen = sizeof(filter);
3 getsockopt(socket_fd, SOL_CAN_RAW, CAN_RAW_FILTER, &filter, &optlen)
4 // CAN filter ID now in filter.can_id
```

Listing 4.11: Example of retrieving a CAN ID from a socket used in the target.

The other values, length and data, are provided by the mutation engine. Having all values present, a CAN(FD) specific frame can now be constructed and written to the memory address pointed to by the paramter *buf*. A simplified example is seen in listing 4.12.

```
1 struct can_frame frame;
2 frame.can_id = used_can_filter.can_id;
3 frame.len = num_copied;  // from mutation engine
4 // data_buffer contains the data provided by the mutation engine
5 memcpy(frame.data, &data_buffer[next_data], num_copied);
```

Listing 4.12: Example of extracting target input if SocketCAN is used.

The decision to limit the mutation engine's involvement to the data section of a CAN message was intentional. By ensuring that the CAN ID and data length are always set

[16] https://man7.org/linux/man-pages/man2/setsockopt.2.html

to their correct values, the risk of false positive encountered crashes is minimized, and the potential for reaching deeper program states during fuzzing is increased.

4.10 Crash Reproduction Workflow

Figure 4.5 illustrates all the steps to manually reproduce a crash found by the fuzzer. The remaining section contains a detailed description of the steps included.

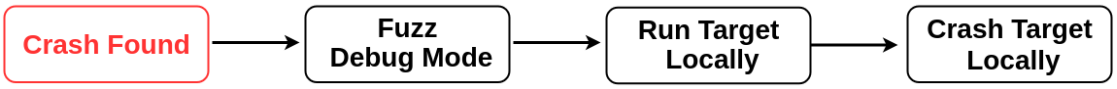

Figure 4.5: Steps to reproduce a crash.

- Crash found: Execute *all_canlinux.sh* to start fuzzing and wait until a crash is found. For a comprehensive description of the fuzzing process itself see section 4.7.
- Fuzz in debug mode: Execute *canreceive_rust_fuzzer_debug.sh* to run the fuzzer in debug mode. This saves the inputs which caused a crash to */tmp/workdir/corpus/crash_reproducible/*.
- Run target locally: Run the target locally listening on CAN interface with name *can0*.
- Crash target locally: This can be done through running the script *can-send-data.py* with the target input as argument:

  packer/packer/can-send-data.py /tmp/workdir/corpus/crash_reproducible/<file name>

can-send-data.py, the file which helps to crash a target locally, is able to operate with input that represents a *can_frame* 3.1 or *canfd_frame* 3.2 message in byte format. In this context a *can_frame* message is always 16 bytes long, with the first 8 bytes representing the CAN ID and the trailing 8 bytes containing the payload data. A *canfd_frame* message is only different in the length of the payload being 64 bytes. The script uses the Python library *python-can* [17] to send a CAN message.

In the current implementation, it is assumed that the CAN filter is set with a mask of *0xFF0*, indicating that the lower 4 bytes of the 8 byte CAN ID are not used. Therefore, only the first four bytes need to be considered when extracting the CAN ID. To obtain the ID used in the target program, the individual bytes are transformed from little-endian to big-endian. The same transformation is done for the payload to get the correct values. Subsequently, a CAN message comprising these two values is transmitted through the CAN interface named *can0*.

[17] `https://python-can.readthedocs.io/en/stable/`

If the target program that encountered the crash, detected by the fuzzer, is configured to receive messages on the *can0* interface, the reproduction of the crash should be achievable with this approach.

The following example should make the workings of *can-send-data.py* more understandable. Considering the bytes *50 05 00 00 08 FE 0C 25 00 06 01 00 FF 08 00 00* as input for the script, the CAN ID is calculated by transforming the first four bytes from little-endian format to big-endian, which results in *05 50* and from the last 8 bytes the payload, respectively the data part of the can_frame 3.1 is calculated the same way and results in *00 00 08 ff 00 01 06 00*. These values are then included in a CAN message and send through the CAN interface to the target program.This takes the input provided by the fuzzer running in debug mode and sends it to the target program.

4.11 Fuzzing Example Server SocketCAN

This section provides a simplified description of reading from the SocketCAN interface, aiming to provide readers with sufficient information to understand the fuzzing process of these programs. Further, it outlines the behaviour of the fuzzer's *Agent* component. It primarily delves into the hooking of specific libc calls. The described logic represents one fuzzing round, which is repeated multiple times during the whole fuzzing process of a target. In subsequent rounds, no new connection is established, but the injected fuzzing input varies from round to round.

The fuzzing targets in this thesis are C programs that communicate with CAN devices using the SocketCAN interface. For communication raw socket protocols are used and the name of the SocketCAN interface is always *can0*. Definitions for the CAN network layer and the raw CAN sockets can be found in the Linux header files linux/can.h [18] and linux/can/raw.h [19]. Comprehensive information about the SocketCAN package is available in the official documentation [20].

To begin communication over a CAN interface, the first step is to open a socket using the *socket*[21] system call. As the raw socket approach is used, the arguments PF_CAN, SOCK_RAW, and CAN_RAW are specified, representing the domain, type of socket, and protocol family, respectively. Example code is shown in listing 4.13.

```
1 int s;
2 s = socket(PF_CAN, SOCK_RAW, CAN_RAW);
```

Listing 4.13: Example of creating CAN socket.

[18] https://github.com/linux-can/can-utils/blob/master/include/linux/can.h

[19] https://github.com/linux-can/can-utils/blob/master/include/linux/can/raw.h

[20] https://docs.kernel.org/networking/can.html

[21] https://man7.org/linux/man-pages/man2/socket.2.html

Next, the created socket binds to the CAN interface. To do this the correct address has to be provided to the *bind* [22] system call. The address family is set to PF_CAN, which corresponds to the socket type. To ensure that the socket is bound to the correct interface, the interface index of the address is set to the CAN interface with the name *can0*. The index is obtained using the *ioctl* [23] system call. Refer to Listing 4.14 for the code example.

```
1 struct sockaddr_can addr;
2 struct ifreq ifr;
3
4 addr.can_family = PF_CAN;
5 strcpy(ifr.ifr_name, "can0");
6 ioctl(s, SIOCGIFINDEX, &ifr);
7 addr.can_ifindex = ifr.ifr_ifindex;
8 bind(s, (struct sockaddr *)&addr, sizeof(addr));
```

Listing 4.14: Example of binding to CAN socket.

It is possible to filter out CAN messages the program receives. This is commonly done in practice but not mandatory. The filtering process occurs at the driver level, providing more efficiency compared to filtering each frame in the user-mode application. To establish a filter, an array of *can_filter* structures is passed to the *setsockopt* [24] system call. A *can_filter* struct consists of a *can_id* and *can_mask* and the filter matches if *<received_can_id> & can_mask == can_id & can_mask*. In the provided code snippet 4.15, the program receives only messages with an CAN ID of 0x550. The CAN mask is set to *0xFF0*, indicating that the lower 4 bytes of the CAN ID are not considered in the filtering process.

```
1 rfilter[0].can_id   = 0x550;
2 rfilter[0].can_mask = 0xFF0;
3 setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, &rfilter, sizeof(rfilter));
```

Listing 4.15: Example of filtering CAN messages.

[22] https://man7.org/linux/man-pages/man2/bind.2.html
[23] https://man7.org/linux/man-pages/man2/ioctl.2.html
[24] https://man7.org/linux/man-pages/man3/setsockopt.3p.html

To receive messages, various functions associated with reading from a socket can be used. In this case, the *read* [25] function is utilised, which writes the received message into a buffer of type *can_frame* 3.1 or *canfd_frame* 3.2. An illustrative code snippet can be found in listing 4.16.

```
1 struct can_frame frame;
2 read(s, &frame, sizeof(frame));
```

Listing 4.16: Example of reading CAN messages.

Upon the invocation of the *bind* function presented in listing 4.14, the *Agent* intercepts the call and performs two actions. Firstly, it invokes the actual *bind* [26] function to complete the socket binding process. Additionally, it prepares the fuzzer for future interactions with this socket. If there is no existing connection to a CAN interface, the *Agent* creates a new connection (see 4.9), adds the target program's socket file descriptor to the *server_sockets* attribute, increments the value of *server_sockets_num* by one, sets *is_can* to true and includes this connection in the list of active connections. Subsequently, all communication of the target program via the SocketCAN interface is managed through this established connection. Assuming the socket file descriptor has value 1, listing 4.17 shows the state of the existing connection after these steps.

```
1 server_sockets = [1]
2 server_sockets_num = 1
3 client_sockets = []
4 client_sockets_num = 0
5 port = <empty>
6 is_can = true
7 disabled = false
```

Listing 4.17: Connection state as described in listing 4.9 after bind hook.

The next call intercepted by the *Agent* is *read* (see listing 4.16). Because there exists a connection to the socket the target program reads from, fuzzing input is injected. This is done the following way: The *frame* argument of the read call is expected to be of type *can_frame* 3.1, Thus, the *Agent* generates a new *can_frame* with the *can_id* set according to the socket options, the *data* field populated with the bytes from the mutation engine, and the *len* field indicating its length. Listing 4.18 shows the values of the frame.

```
1 can_id = 0x550
2 len = 8
3 data = 0102030405060708
```

Listing 4.18: CAN frame values which are injected into target program.

[25] https://man7.org/linux/man-pages/man2/read.2.html
[26] https://man7.org/linux/man-pages/man2/bind.2.html

This newly created *can_frame* is then written to the memory address pointed to by the *frame* parameter. The function returns the size of the created *can_frame*, or -1 if an error occurs during execution.

Following the call to *read*, the target program would proceed to handle the received input stored in the *frame* variable. If the target program closes the socket, the connection created in the *Agent* component is deleted.

CHAPTER 5

Evaluation

This chapter undertakes the evaluation of the prototype fuzzer formulated and developed as a part of this thesis. Evaluation serves as the process to assess the extent to which the designed solution aligns with its intended objectives and successfully meets requirements.

The chapter starts by outlining the specifics of the evaluation setup, followed by the evaluation of the fuzzer Nyx-Net (without changes). Prior to delving into the assessment of test cases affiliated with SocketCAN, a representative target program that employs TCP sockets is evaluated in section 5.3. Following this preliminary evaluation, the focus of assessment pivots towards targets reliant on the SocketCAN protocol. The evaluation encompasses a spectrum of target programs, characterized by varying behaviours and degrees of complexity, aiming to provide a comprehensive insight into the fuzzer's capabilities.

5.1 Setup

The prototype fuzzer is developed and tested on a laptop possessing the following specifications:

- **Memory**: 16 GiB
- **Processor**: 12th Gen Intel Core i7-1260P × 16
- **OS**: Ubuntu 22.04.2 LTS
- **Linux Kernel**: 5.19.0-50-generic

Fuzzer configuration

There are three ways of fine-tuning the fuzzers configuration, snapshot scheduling, parallel fuzzing and adaption of the target's input specification:

- Snapshot scheduling: The fuzzer offers the flexibility of selecting the point at which an incremental snapshot is taken, this feature is already integrated into Nyx-Net and is not changed in any way for this thesis' implementation. Three snapshot scheduling strategies are available: *Nyx-Net-none*, *Nyx-Net-balanced*, and *Nyx-Net-aggressive*. In the *Nyx-Net-none* strategy, only the root snapshot is chosen, and no incremental snapshots are taken. With the *Nyx-Net-balanced* strategy, inputs containing more than four packets select the root snapshot 4% of the time, while others randomly choose an index in either the entire sequence or the second half. The *Nyx-Net-aggressive* strategy initiates snapshots at the end of an input the first time, then progressively shifts the snapshot earlier by one packet every 50 iterations if no new inputs are discovered. Once the earliest index is reached, the process restarts from the input's end. This dynamic approach allows the fuzzer to exploit incremental snapshots more effectively, considering a balance between snapshot timing and the exploration of different paths in the sequence.

- Parallel fuzzing: This can be activated by utilizing the *-t* parameter when initiating the fuzzer. This parameter specifies the number of concurrent running fuzzer instances.

- Adapt target input specification: As outlined in section 4.8, the provision of seed inputs can influence the fuzzer's behaviour. When these seed inputs are closely related to potential crash-triggering inputs, there's a higher probability that the fuzzer will generate such inputs. Another way is to establish rules for the fuzzing input using regular expression. However, this thesis doesn't delve into this method, because as indicated in the Nyx-Net paper, fast performance is attainable by solely providing seed inputs without requiring an intricate protocol-specific description.

5.2 Evaluation of Setup

Prior to starting the creation of the new prototype fuzzer capable of fuzzing targets utilising SocketCAN, it is imperative to complete the setup of Nyx-Net. To validate if the fuzzer is able to run, the target example dnsMasq, which is already included in the Nyx-Net fuzzer, is fuzz tested.

Following the adjustments outlined in section 4.2, it becomes feasible to conduct fuzzing on dnsMasq. The testing session was executed for a duration of 20 minutes, which was to short to find crashes. To ensure the continued effectiveness of the fuzzer after implementing modifications, a set of simpler test targets with introduced vulnerabilities are created. Detailed descriptions of these targets can be found in both Section 5.3 and

Section 5.5. These targets are structured in a manner that facilitates the fuzzer's ability to uncover program crashes within a reasonable amount of time.

One of the initial modifications implemented for the new fuzzer involved replacing the operating system utilised by the virtual machine in which the *Agent* component and target program are executed. The details of these adjustments are elaborated upon in section 4.4. Although it is feasible to manually revert to the previous operating system configuration, the aim is to ensure users' convenience by enabling fuzzing of classical socket targets using the new minimal Linux operating system. To validate this, the target dnsMasq was also subjected to fuzz testing with the new operating system, resulting in no identified issues.

5.3 Evaluation without SocketCAN

To ensure the new fuzzer's compatibility with targets that do not rely on the SocketCAN interface for data reception and to verify the fuzzer's ability to find crashes, a basic target program is developed and evaluated. This program is available in the *additional-files/customsocket* directory and is coded in the C programming language. The intention behind its creation is to emulate a conventional TCP server program and simulate a vulnerability. The program is structured as outlined below:

1. Create a TCP socket.
2. Bind socket to the server address.
3. Wait for the client to make a connection.
4. Establish a connection to the client.
5. Receive data on the socket.
6. If the received data possess a certain condition, crash the program. A crash is simulated with a call to *abort* [1].

The crash conditions are simple checks for certain bytes in the received data of the target program. The fuzzer is able to identify this vulnerability in approximately 10 seconds, enabling the verification of the fuzzer during development through fuzz testing on this target.

5.4 Evaluation of Input Specification on SocketCAN Targets

To verify the suitability of the input specification tailored for SocketCAN targets, the newly introduced CAN_SPEC_TEST_MODE can be employed. A comprehensive

[1] `https://man7.org/linux/man-pages/man3/abort.3.html`

explanation can be found in section 4.6. The suitability criteria for data which is created by using the SocketCAN input specification are:

- Numerous input messages must be injected into the target.
- The data created must differ between each other. While it's acceptable for the same data to occur multiple times, the fuzzer should avoid consistently producing identical data.
- There should be multiple messages injected into the target per second.

Executing the experiment via the *all_can_spec_test.sh* script shows the fulfilment of all outlined requirements. A sample output of the *candump* tool is presented in Listing 5.1, where the columns signify: the Unix timestamp [2] in seconds, the interface name of message reception, the CAN ID of the message, the data length, and the payload data of the transmitted message. The fuzzer takes around 10 to 15 seconds to create other payload data than zero bytes but after that all the requirements are satisfied. As shown, multiple messages are getting sent to the target program. It is not seen in the output but there are sometimes messages which are having the same payload. This only occurs for a time frame of about 2 to 3 seconds. In general, the payload differs quit broad. As seen on the timestamp in the left most column, there are also multiple messages send from the fuzzer to the target program within one second.

```
1   (1692290616.276581)   can0   000   [8]   00 00 00 00 00 00 00 00
2   (1692290616.276623)   can0   000   [8]   00 00 00 00 00 00 00 00
3   (1692290616.276665)   can0   000   [8]   8E E3 03 A6 ED 57 8F F8
4   (1692290616.276711)   can0   000   [8]   6E B2 E3 56 DE 25 A4 A1
5   (1692290616.276757)   can0   000   [8]   9D B1 5E 33 F6 B9 63 0E
6   (1692290616.276802)   can0   000   [8]   E7 06 31 05 2D 55 EB 93
7   (1692290616.276847)   can0   000   [8]   13 19 0D A4 59 8E F9 87
8   (1692290616.276892)   can0   000   [8]   BD 5F 11 75 36 BD F1 56
9   (1692290616.276938)   can0   000   [8]   25 26 D4 5C E2 BA 60 DB
10  (1692290616.276983)   can0   000   [8]   09 44 B9 1B C5 64 8C E5
11  (1692290616.277028)   can0   000   [8]   70 97 8F C3 4A 8D 31 2C
```

Listing 5.1: Snapshot of target input while fuzzing.

5.5 Evaluation of SocketCAN Targets

To assess the prototype fuzzer's performance with SocketCAN targets, C programs are used as the target applications. These programs intentionally contain vulnerabilities, and the fuzzer's capability to discover them is evaluated. The target applications can be categorized into two groups: stateful and stateless.

[2] https://www.unixtimestamp.com/

5.5.1 Stateless Target

Different examples of stateless targets can be found in *additional-files/0CanExamples*. These examples all share a common underlying principle:

1. Bind socket to a CAN/CAN FD interface.
2. Optional: set filter for a CAN ID.
3. Use read(), recv() or recvfrom() to receive data.
4. If the received message contains certain bytes, crash the program. For example, the program crashes if the message received comes with ID 0x550 and the data payload includes the bytes 06 and FF.

The messages received are in the classical CAN or in the CAN FD format, meaning they have the form of a can_frame 3.1 respectively canfd_frame 3.2. A crash is simulated with a call to *abort* [3], this causes an abnormal process termination which is detected and recorded by the fuzzer. To trigger a crash the fuzzer must send a payload which contains specified bytes. Listing 5.2 shows how crashes are triggered in target programs. In the given example a crash is triggered if the payload of the message contains the byte 06 and FF. The byte comparison could be written within only one if-block, but by breaking down the comparisons into nested steps, the fuzzer can gradually piece together the "puzzle" and determine when it's on the correct path, typically indicated by reaching a deeper program state. This approach is called *comparison coverage* and was first introduced within the fuzzer AFL++ [34]. Some compilers used for fuzzing are applying such techniques automatically. For example, the additional LLVM passes implemented for the AFL fuzzer have integrated solutions to create comparisons in this form. These passes assist in systematically exploring program execution paths and identifying promising directions for further exploration. The target programs used for these test runs also employ a filtering mechanism based on specific CAN IDs. As this is done through the socket option configurations, it is not re-verified whether the target program exclusively crashes when a particular CAN ID is configured.

```
1 if(containsByte(&frame, 0x06)) {
2     if(containsByte(&frame, 0xFF)) {
3         abort();
4     }
5 }
```

Listing 5.2: Example crash trigger in target programs.

[3] https://man7.org/linux/man-pages/man3/abort.3.html

can-test

The target programs utilised for this section are drawn from the *can-test* GitHub repository [4], which is curated and managed by the developers of SocketCAN. This repository contains a collection of diverse SocketCAN test applications. These programs can serve as suitable fuzzing targets with just a minor adjustment. The user interactions are omitted to facilitate direct submission of fuzzing input by the fuzzer as soon as the target program executes. Assessment runs encompass different levels of difficulty to identify crashes. As the test targets are quit simple, these test cases are specifically employed to verify the fuzzer's capability to uncover crashes and do not significantly reflect its performance on more complex targets. The provided times aim to offer readers an insight into the complexity of the individual test scenarios.

Crash Values	Seconds to Crash (CAN)	Seconds to Crash (CAN FD)
C3	0.1	0.19
DA	0.25	0.53
06, FF	0.2	0.26
11, AD	0.82	0.95
06, FF, 11	0.92	0.28
DD, AC, BC	0.88	1.42
00, 04, E3, 84	78.76	0.29
C3, F5, 09, DA	10.11	0.14
01, 02, 03, 04, 05	2.09	0.13
32, 5F, 1A, 52, E4	435.62	4.09

Table 5.1: Fuzzer evaluation collection with read call and filter for ID.

There exist two test targets utilising the read [5] function to receive messages. One target employs classical CAN communication, while the other utilises CAN FD. Both programs only accept messages with a CAN ID of 0x550.

Table 5.1 shows the evaluation of different test runs. The *Crash Values* column illustrates the bytes necessary in the payload of the CAN message to induce a crash. The two other columns indicate the time in seconds taken by the fuzzer to detect a crash, once for conventional CAN messages and again for CAN FD messages.

Examining table 5.1, it is evident that when communication is conducted through CAN FD, the fuzzer can detect crashes faster as the test case difficulty increase. This phenomenon occurs because classical CAN messages have a payload length of only 8 bytes, while CAN FD messages consist of 64 bytes. Thus, the fuzzer creates different inputs having a maximum length of 8 bytes respectively 64 bytes. By introducing crashes in target programs as illustrated in listing 5.2, the likelihood of finding a specific byte in the longer messages is higher. This is the case because the fuzzer is able to send more

[4] https://github.com/linux-can/can-tests

[5] https://man7.org/linux/man-pages/man2/read.2.html

distinct bytes within one fuzzing input. Furthermore, these test runs demonstrate that the fuzzer can successfully infer more than 50% of a CAN message data bytes within a time frame of 7.5 minutes.

To reproduce these test runs, use the file *additional-files/0CanExamples/can_read_filter.c* for CAN messages and *additional-files/0CanExamples/can_fd_read_filter.c* for CAN FD messages. Adapt the abort condition as needed, copy the content into *additional-files/canLinuxExample/canreceive.c* and run *all_canlinux.sh*. Crash times are written to */tmp/workdir/crash_times.txt*.

The next target uses the recv [6] function for communication and filters for 0x550 as CAN ID. The message received is in the form of a classical CAN frame.

Outcomes of the test runs can be seen in table 5.2. The data shows that the fuzzer is able to infer up to 5 bytes of the 8 bytes message payload within two minutes. Particularly, the test runs involving *06, FF; 06, FF, 11, and DD, AC, BC* exhibit rapid crash discovery by the fuzzer. This could be attributed to the inclusion of these values in the initial seed inputs, prompting the fuzzer to prioritize them in the initial fuzzing iterations.

Crash Values	Seconds to Crash (CAN)
C3	0.12
DA	3.01
06, FF	0.0025
11, AD	0.39
06, FF, 11	0.002
DD, AC, BC	0.002
00, 04, E3, 84	8.96
C3, F5, 09, DA	118.35
01, 02, 03, 04, 05	1.9
32, 5F, 1A, 52, E4	115.75

Table 5.2: Fuzzer evaluation collection with recv call and filter for ID.

To reproduce these test runs use *additional-files/0CanExamples/can_recv_filter.c*. Copy the content to *additional-files/canLinuxExample/canreceive.c*, adapt the abort condition as needed, and run *all_canlinux.sh*. Crash times are written to */tmp/workdir/crash_times.txt*.

[6] `https://man7.org/linux/man-pages/man2/recv.2.html`

Another target used for verification is utilising the function recvfrom [7] and does not filter for messages having a specific CAN ID. The type of the communication is classical CAN.

Table 5.3 demonstrates the outcome of the individual test runs. In general, it's evident that the time required to discover the initial crash falls within the range of approximately 6.5 minutes. With test runs having three or less crash values being much quicker. It's noteworthy that as the complexity of the test cases increases, the time to detect a crash also extends, with one notable exception being the test run featuring the crash values *01, 02, 03, 04, 05*. This anomaly might be attributed, once more, to these values being part of the initial seed inputs, causing the fuzzer to quickly identify them in the initial stages.

Crash Values	Seconds to Crash (CAN)
C3	1.1
DA	0.9
06, FF	1.71
11, AD	1.7
06, FF, 11	1.9
DD, AC, BC	1.7
00, 04, E3, 84	126.89
C3, F5, 09, DA	375.49
01, 02, 03, 04, 05	3.99
32, 5F, 1A, 52, E4	57.14

Table 5.3: Fuzzer evaluation collection with recvfrom call.

To reproduce these test runs use *additional-files/0CanExamples/can_recvfrom_no_filter.c*. Copy the content to *additional-files/canLinuxExample/canreceive.c*, adapt the abort condition as needed, and run *all_canlinux.sh*. Crash times are written to */tmp/-workdir/crash_times.txt*.

[7] https://man7.org/linux/man-pages/man2/recv.2.html

The experiments presented in this section provide evidence of the fuzzer's effectiveness on target programs that employ the SocketCAN interface. Moreover, the fuzzer demonstrates its capability to manage different target implementations.

A comparison between experiments done so far (tables 5.1, 5.2, 5.3) is shown in figure 5.1. The mere utilization of distinct functions (read, recv, recvfrom) for message reception shouldn't inherently result in the observed variations in overall time. This is because recv and recvfrom are employed without any additional option flags, and as such, their behavior should be analogous to that of read. However, these time differences can likely be attributed to two main factors. Firstly, the fuzzer doesn't apply the same mutations to the input in every run, leading to variable execution times. Secondly, the target programs vary for each type of message reception, leading to a different position of the crash condition.

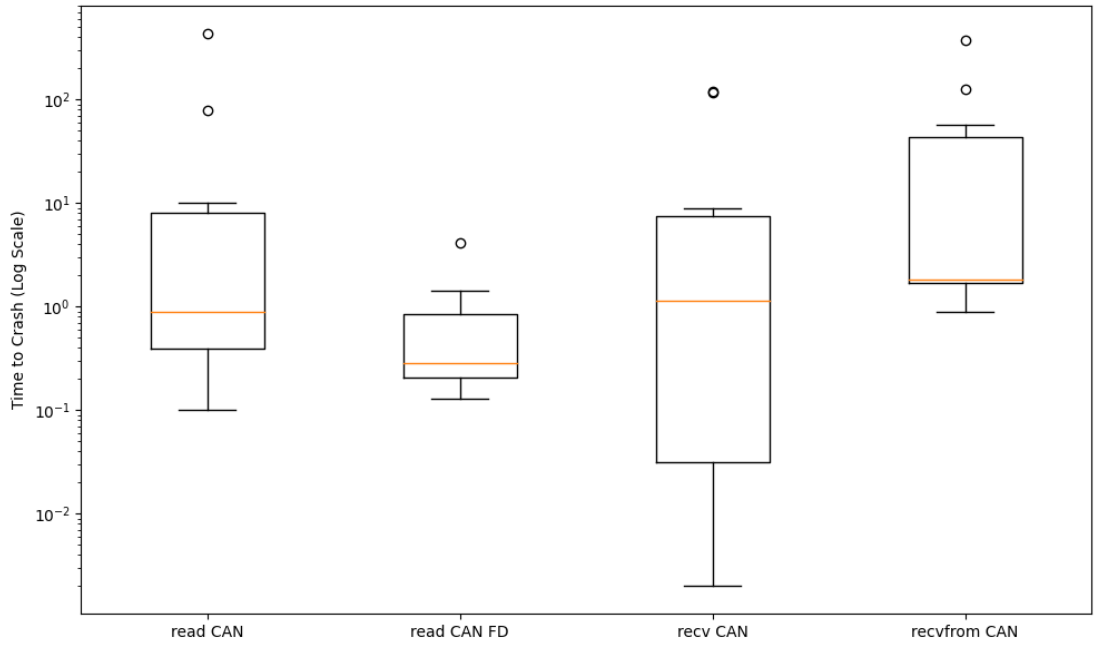

Figure 5.1: Time to crash comparison for target programs utilising read, recv and recvfrom.

When comparing the medians of the various experiments, it becomes evident that *read CAN FD* tends to have lower values. This further supports the hypothesis that when searching for a byte within a collection of 64 bytes, it is easier for the fuzzer compared to searching within a collection of 8 bytes.

The observed times for experiment *recv CAN* display the greatest degree scatter. This variability can be attributed, again, to the initial fuzzing values employed by the fuzzer. At certain points, the fuzzer progressed rapidly, as evidenced by test runs completing

within 0.002 seconds to cause a crash. Conversely, there were instances where the fuzzer required significantly more time, such as the test run that consumed 118.35 seconds.

It is also worth noting that all the outliers fall above the upper whiskers. This suggests that the majority of values are on the smaller side, but there are a few instances with exceptionally large values. This phenomenon is reinforced by the right skewness apparent in most of the experiments.

Another stateless test scenario introduces a more practical vulnerability: Upon data reception, a mathematical computation is performed to determine an index for accessing an array. This scenario replicates an out-of-bounds read vulnerability [8], where data is read beyond the intended buffer's end or before its beginning. Following data reading, the index calculation is based on the values of byte 1 added to byte 2 minus byte 3 of the received payload. If the calculated index is less than zero or greater than the array's length, which results in accessing a memory location beyond the array's bounds, an exceptional process termination occurs. Executing this fuzz test case leads to the discovery of a crash in 0.26 seconds. The introduced vulnerability can be seen in listing 5.3. The parameter *frame* is provided by the fuzzer.

Demonstration of the out-of-bounds read illustrates that the fuzzer is effective for more practical examples as well. While the logic to trigger a crash isn't overly complex, it highlights the fuzzer's ability to detect this type of bug. As the difficulty level increases, it's expected to impact only the time it takes the fuzzer to identify the crash, rather than its ability to find one.

```
1 int accessArrayOnIndex(const struct can_frame *frame) {
2     int myNumbers[4] = {25, 50, 75, 100};
3     int index = frame->data[0] + frame->data[1] - frame->data[2];
4     int value = myNumbers[index];
5     return value;
6 }
```

Listing 5.3: Out-of-bounds read vulnerability.

[8] https://cwe.mitre.org/data/definitions/125.html

can-utils

The project can-utils [9] is a composition of SocketCAN user space applications. It contains utilities for:

- Display, record, generate and replay CAN traffic
- CAN access via IP sockets
- CAN in-kernel gateway configuration
- CAN bus measurement and testing
- ISO-TP tools (implements data transfers according ISO 15765-2)
- J1939/ISOBus tools
- Log file converters (e.g. convert ASC logfile to compact CAN frame logfile)
- Serial Line Discipline configuration (e.g userspace tool for serial line CAN interface configuration)

Included in the toolset is a CAN message logging server tool named *canlogserver*. This utility serves as a practical example for testing the fuzzer. To ensure reproducibility and eliminate dependency on changes in the GitHub repository, a self-contained copy of the program and its dependencies are stored within the directory named *additional-files/canlogserver*. Each file is accompanied by the corresponding SHA commit hash associated with the last modification. The complete project encompasses:

- The CAN log server: canlogserver.c, including a vulnerability which is triggered if the received CAN message includes certain data bytes.
- The file to build the log server: Makefile
- Dependencies and helpers: include, check_cc.sh, fork_test.c, libc. lib.h

During the test runs, the logging server is configured to listen to all accessible CAN interfaces. Messages are transmitted via a CAN interface named can0.

[9] `https://github.com/linux-can/can-utils`

A vulnerability is introduced in the program the following way: Upon receiving a message, the logserver crafts a log message that encapsulates key details of the received input, such as its reception time and size. During this log message construction, an offset is deduced. The offset calculation is the helper function showcased in listing 5.4. The primary role of this function is to fetch an alternate CAN FD frame to determine the offset within the log message string buffer. Notably, there's an instance where this function returns a NULL pointer. Subsequently, an attempt to read from the intended CAN FD frame inadvertently dereferences this NULL pointer.

```
1  struct canfd_frame* get_related_frame(struct canfd_frame *cf) {
2      // Some (contrived) condition based on the data that
3      // determines if NULL should be returned.
4      if (cf->data[4] == 0xAA) {
5          if (cf->data[5] == 0xBB) {
6              return NULL;
7          }
8      }
9
10     // In a real-world scenario, there would be some logic
11     // here to get a related frame.
12     // For our purposes, we'll just return the input frame.
13     return cf;
14 }
```

Listing 5.4: Introduced vulnerability in canlogserver target.

The fuzzer can identify this kind of vulnerability in less than one second. This demonstrates the fuzzer's effective capability to detect NULL pointer vulnerabilities.

To replicate the test scenarios, modify the crash condition within the file *additional-files/canlogserver/canlogserver.c*, and then run the script *all_canlogserver.sh*. The recorded crash detection times are saved in the file */tmp/workdir/crash_times.txt*.

Another program included in the can-utils project is *cansniffer*. It is a command-line utility commonly used in CAN bus analysis. It allows to monitor and capture CAN traffic on a network. By listening to data frames transmitted on the CAN interface, *cansniffer* provides insights into the communication between different nodes, for example to identify messages, their IDs, or payload content in real-time.

Similar to the *canlogserver* example, to guarantee reproducibility and remove reliance on alterations in the GitHub repository, a duplicate of the program along with its necessary components is stored in the directory denoted as *additional-files/cansniffer*, which consists of:

- The CAN sniffer program: cansniffer.c, including a vulnerability
- The file to build the log server: Makefile

- A C header file used by the sniffer: terminal.h

Following conditions need to be met to cause the program to crash:

- The payload data must be different from the previous message with the same CAN ID
- For the classical CAN message format, the received data length must match the size of a can_frame 3.1, and for CAN FD messages, it must correspond to the size of a canfd_frame 3.2
- The program *cansniffer* can only manage up to 2048 distinct (by CAN ID) messages simultaneously; if there are more messages, the program won't accept new ones; the stored messages are automatically cleared after a specific time duration

A buffer overflow occurs when all the aforementioned conditions are satisfied. It is introduced in the following manner: The function depicted in listing 5.5 attempts to swap the data values of two distinct *canfd_frames*. The issue arises because the temporary variable, intended to aid in swapping the values, is of type *can_frame* and thus has a smaller *data* attribute size. Consequently, when executing line 3 of the code snippet, arbitrary memory is overwritten. This problem is compounded by the usage of the function *returnChunkSize()*, which is responsible for determining the correct number of bytes to be copied. The problem is that it has a flaw in its calculation, for example it can return -1, causing *memcpy* [10] to attempt copying the maximum number of bytes.

```
1 int vulnerable_function(struct snif input) {
2     struct can_frame tmp;
3     memcpy(tmp.data, input.marker.data, returnChunkSize());
4     printf("tmp data: %s", tmp.data);
5     memcpy(input.marker.data, input.notch.data, returnChunkSize());
6     memcpy(input.notch.data, tmp.data, returnChunkSize());
7     return 0;
8 }
```

Listing 5.5: Introduced vulnerability in cansniffer target.

The fuzzer is able to detect the crash in 1.3 seconds. This shows that it effectively navigates through the program's deeper states without encountering issues. The conditions required to reach these deeper program states are met without substantial time discrepancies compared to the other test runs in previous sections.

To reproduce the test cases, adapt the crash condition inside *additional-files/cansniffer/cansniffer.c* and execute the file *all_cansniffer.sh*. The times for detecting a crash are written to */tmp/workdir/crash_times.txt*.

[10] https://man7.org/linux/man-pages/man3/memcpy.3.html

5.5.2 Stateful Target

The fuzzer implemented in this thesis employs incremental snapshots during the fuzzing process, making it particularly effective for fuzzing stateful target programs. This section provides an evaluation of using the fuzzer with such a stateful target.

All bytes within the CAN FD message payload are presented in hexadecimal format.

The full target program can be found under *additional-files/canvehicle/canvehicle.c*.

This test target undergoes fuzz testing using two different tools. Firstly, the fuzzer developed within this thesis is employed. Secondly, cangen [11], a tool for generating random CAN frames and transmitting them through a CAN interface, is utilised.

The target program simulates various vehicle components. The fuzzer and other programs are able to communication with the target via the SocketCAN interface with name *can0*. Messages exchanged via this interface must be assigned a CAN ID of 0x550 and adhere to the extended CAN FD format.

Every component inside the program has a certain state. For most of them the state can be on and off, but there is one component called *speed* which consists of an integer representing the speed in km/h. Following components exists:

- front light (on/off)
- heating (on/off)
- aircon (on/off)
- privileged mode (on/off)
- speed (integer)

Furthermore, there are actions that can be initiated, all of which are outlined below. Some requiring privileged mode for execution.

- turn components on or off
- password check to go in privileged mode
- calculate speed of the vehicle
- change the engine configuration (only in privileged mode)
- buy a product (only in privileged mode)
- play music (only in privileged mode)

[11] `https://manpages.debian.org/stretch-backports/can-utils/cangen.1.en.html`

The program executes on behalf of the CAN message payload. As the message is in CAN FD format the payload has a maximum length of 64 bytes. To guide the program in different directions, rules are established on behalf of the received payload:

- If the payload is only zeros, the program prints the state of all components.
- If the first two bytes are not zero, they are used to turn the various components on and off.
- If the first two bytes are zero, different actions can be triggered. The specific action depends on the remaining content.

Inside the program five vulnerabilities are introduced.

Out-of-bounds write: An out-of-bounds write vulnerability [12] involves the writing of data beyond the boundaries of a buffer, either before its start or after its end. This often leads to data corruption, program crashes, or even unauthorized code execution. It occurs when the program alters an index or conducts pointer calculations that reference a memory location outside the buffer's defined limits. Subsequent write operations in such cases yield undefined or unexpected outcomes.

Two vulnerabilities of this type are presented in the target program. Listing 5.6 depicts the first. This vulnerability resides within the engine code section, necessitating the program to be executed in privileged mode. Additional crash conditions are: the third byte of the payload must be set to 33, and the fourth byte to 44. Additionally, as evident in the if-statements within the code, the front light and air conditioning must both be turned on. The target program crashes when the expression *distances[NUM_NODES - 1] - 2* results in a negative number. This occurs because the memcpy function [13] assumes that the value is unsigned, leading it to interpret it as the maximum integer value minus one. Consequently, it copies significantly more memory than is available to the destination buffer. The value *distances[NUM_NODES - 1] - 2* is the outcome of a computation based on the Dijkstra shortest path algorithm, dependent on the input received from the fuzzer.

```
1 if (vehicle_state.front_light_state == 1) {
2     if (vehicle_state.aircon == 1) {
3         char destBuf[10];
4         char srcBuf[10];
5         memcpy(destBuf, srcBuf, (distances[NUM_NODES - 1]-2));
6     }
7 }
```

Listing 5.6: First out-of-bounds write vulnerability in stateful target program.

[12] https://cwe.mitre.org/data/definitions/787.html
[13] https://man7.org/linux/man-pages/man3/memcpy.3.html

The second out-of-bounds write vulnerability within the target program is illustrated in listing 5.7. To exploit this vulnerability, privileged mode must be enabled, the third byte of the payload should be set to 55, and the fourth byte must be greater than EC. The program consistently crashes when line 3 of the listing, specifically the statement *memcpy(product2, newProduct, (calculate()-1));*, is executed. This is due to the fact that the function *calculate* consistently returns -1. As a result, the same issue as previously described occurs, where significantly more memory is copied into the destination buffer than it can accommodate.

```
1 if (frame.data[3] >= 0xEC) {
2     char *newProduct[3] = {"N", "E", "W"};
3     memcpy(product2, newProduct, (calculate()-1));
4 }
```

Listing 5.7: Second out-of-bounds write vulnerability in stateful target program.

One potential mitigation strategy for these vulnerabilities is to validate the third parameter of the memcpy call, ensuring that the correct amount of memory is copied.

Out-of-bounds read: In an out-of-bounds read vulnerability [14], a program reads data beyond the end or before the beginning of the intended buffer. Typically, this can enable attackers to access sensitive information from other memory locations or trigger a program crash. A crash may occur when the code reads a variable amount of data and assumes the presence of a sentinel to terminate the read operation, such as a NULL character in a string. However, the expected sentinel might not be located in the out-of-bounds memory, resulting in the reading of excessive data, which can lead to a segmentation fault or buffer overflow. This type of vulnerability can occur when the program modifies an index or performs pointer arithmetic that references a memory location outside the buffer's boundaries, leading to undefined or unexpected results.

The introduced vulnerability in the target program is depicted in listing 5.8. To access this code, the program must operate in privileged mode, the third byte of the payload must be 55, and the fourth byte should be smaller than 33. The *songs* struct contains five entries, thus if *choice - 1* exceeds 4, it results in reading from an undefined memory location. The variable *choice* is influenced by the input provided to the target program where it undergoes a mathematical computation before its usage.

```
1 if (choice > 0) {
2     printf("Now␣playing␣%s\n\n", songs[choice - 1]);
3 }
```

Listing 5.8: Out-of-bounds read vulnerability in stateful target program.

To address this problem, it's advisable to validate the variable *choice* to ensure it doesn't exceed the size of the *songs* struct, rather than solely checking that it's greater than 0.

[14] https://cwe.mitre.org/data/definitions/125.html

NULL Pointer Dereference: A NULL pointer dereference [15] occurs when the application dereferences a pointer that it expects to be valid, but is NULL, typically causing a crash or exit. NULL pointer dereference issues can occur through a number of flaws, including race conditions, and simple programming omissions.

In listing 5.9 the vulnerability inside the target program can be seen. This function is called when privileged mode is enabled, byte number three is 33, byte number four is 44, byte number ten is A4, byte number eleven is 7E and byte number fourteen is A4. If the code is executed it will always end in a crash because the address used as first parameter to *gethostbyaddr* [16] are bytes used from the received message payload of the CAN FD messages. These are never result in an well-formed internet address, leading to *gethostbyaddr* returning NULL. Since the code does not check the return value from *gethostbyaddr*, a NULL pointer dereference occurs in the call to *strcpy* [17]. This scenario may not be very common within a system employing the CAN protocol, but it showcases the fuzzer's capability to handle more intricate input protocol requirements.

```
1  void vulnerable_function(char *user_supplied_addr){
2      struct hostent *hp;
3      in_addr_t *addr;
4      char hostname[64];
5      in_addr_t inet_addr(const char *cp);
6
7      addr = inet_addr(user_supplied_addr);
8      hp = gethostbyaddr(addr, sizeof(struct in_addr),
9                         AF_INET);
10     strcpy(hostname, hp->h_name);
11 }
```

Listing 5.9: NULL pointer dereference vulnerability in stateful target program.

To mitigate this issue, it is important to validate the user input in being a well-formed internet address.

[15] https://cwe.mitre.org/data/definitions/476.html
[16] https://linux.die.net/man/3/gethostbyaddr
[17] https://man7.org/linux/man-pages/man3/strcpy.3.html

Call to abort: The last vulnerability introduced in the target is a call to *abort* [18].

The relevant code piece is shown in listing 5.10. The code piece is reached if the first five bytes of the payload are 00 00 01 55 FF. And as seen in the if-statements the aircon, heating and front light must be turned on to trigger the crash through the call to abort.

```
1 if (vehicle_state.aircon) {
2     if (!vehicle_state.heating) {
3         if (vehicle_state.front_light_state) {
4             abort();
5         }
6     }
7 }
```

Listing 5.10: Abort call in stateful target program.

[18] https://man7.org/linux/man-pages/man3/abort.3.html

Test run evaluation

Table 5.4 shows vulnerabilities found by each test run. The five different vulnerabilities are defined as V1 to V5 and the reference to the listing describing the type is given. The runs are conducted for 5, 15, 30 and 60 minutes with the tool cangen and the prototype fuzzer. For all runs, three fuzzing instances run simultaneously and the snapshot strategies none, balanced and aggressive are used.

Test Run	V1 (lst.5.6)	V2 (lst.5.7)	V3 (lst.5.8)	V4 (lst.5.9)	V5 (lst.5.10)
cangen 5	-	-	-	-	-
cangen 15	-	-	-	-	-
cangen 30	-	-	-	-	-
cangen 60	-	-	-	-	-
none 5	-	-	-	-	-
none 15	✓	✓	✓	-	✓
none 30	✓	✓	✓	-	✓
none 60	-	✓	✓	-	✓
balanced 5	-	-	✓	-	✓
balanced 15	-	✓	✓	-	✓
balanced 30	✓	✓	✓	-	✓
balanced 60	✓	✓	✓	-	✓
aggressive 5	✓	✓	✓	-	✓
aggressive 15	✓	✓	✓	-	✓
aggressive 30	✓	✓	✓	-	✓
aggressive 60	✓	✓	✓	-	✓

Table 5.4: Vulnerabilities found within each test run for the stateful target program. Test runs are executed for 5, 15, 30 and 60 minutes by the tool cangen and the prototype fuzzer with snapshot scheduling of none, balanced and aggressive. For all runs, three fuzzing instances run simultaneously. The specific vulnerabilities are defined as V1 to V5 with the description reference appended.

The performance of the prototype fuzzer significantly outperforms the cangen tool. Notably, cangen fails to uncover any vulnerabilities, whereas the prototype fuzzer successfully detects all vulnerabilities except for V4, which relates to a NULL pointer dereference.

Additionally, the impact of the fuzzer's scheduling configuration is evident. Without utilising any intermediate snapshots (none), the fuzzer takes more time to uncover vulnerabilities, and in some instances, such as V1, it fails to do so even after running for 60 minutes. When applying balanced snapshot scheduling, the fuzzer encounters difficulties in finding V1 but eventually succeeds given enough time.

The most consistent and reliable results are achieved when the prototype fuzzer operates in aggressive snapshot scheduling mode. In all test runs conducted under this configuration, vulnerabilities V1, V2, V3, and V5 are successfully detected. As previously mentioned,

V4 remains elusive, but given the fuzzer's increasing line and branch coverage over time, it is reasonable to assume that it will be discovered with extended fuzzing runtime.

Table 5.5 provides a comprehensive summary of the line coverage statistics across all test runs. For a clearer representation, please refer to the line graph illustrated in Figure 5.2.

Test Run	Lines Total	Lines Covered	Lines Covered (%)
cangen 5	175	45	0.26
cangen 15	175	56	0.32
cangen 30	175	56	0.32
cangen 60	175	56	0.32
none 5	175	80	0.46
none 15	175	155	0.89
none 30	175	150	0.86
none 60	175	152	0.87
balanced 5	175	105	0.60
balanced 15	175	152	0.87
balanced 30	175	161	0.92
balanced 60	175	159	0.91
aggressive 5	175	161	0.92
aggressive 15	175	152	0.87
aggressive 30	175	157	0.90
aggressive 60	175	159	0.91

Table 5.5: Line coverage within each test run of the stateful target. Test runs are executed for 5, 15, 30 and 60 minutes by the tool cangen and the prototype fuzzer with snapshot scheduling of none, balanced and aggressive. For all runs, three fuzzing instances run simultaneously

When comparing the performance of the prototype fuzzer with the tool cangen, it becomes evident again that the fuzzer consistently outperforms cangen. Even after running for a full 60 minutes, cangen is unable to achieve line coverage beyond 32%.

Using the fuzzer without intermediate snapshots proves to be less efficient compared to the balanced or aggressive scheduling. The test run for 5 minutes achieves a line coverage of only 39%, indicating that it takes longer to explore program lines. Similarly, after 60 minutes, the line coverage reaches only 87%, highlighting that not employing intermediate snapshots is not as effective as the other options in a longer time period.

The fuzzer's balanced snapshot scheduling option achieves higher line coverage after 5 minutes compared to the none option. It even surpasses the line coverage of the aggressive scheduling option within the 30 minute test run.

An interesting observation emerges regarding the aggressive snapshot scheduling option. In this case, the test run lasting only 5 minutes outperforms all other runs, achieving a 92% line coverage. This highlights the variability in the fuzzer's performance within

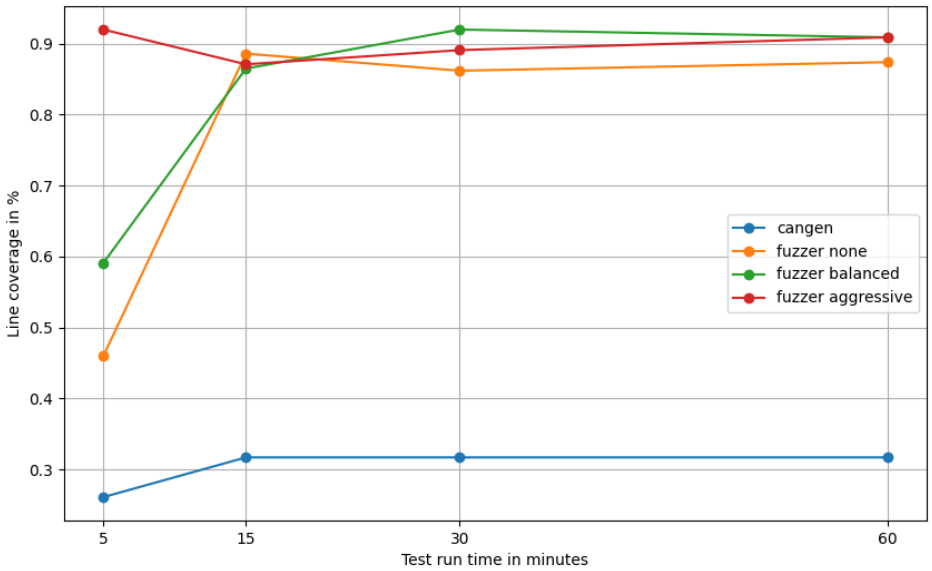

Figure 5.2: Line coverage comparison of each test run of the stateful target.

different test runs on the same target. The variation is mainly attributed to the initial selection of highly suitable inputs. However, when examining the line coverage with the none and balanced options, it becomes also apparent that the fuzzer may begin with less suitable inputs but progressively improves its performance over time to yield favorable results.

Within a computer program, a branch refers to one of the potential execution paths the code may follow during the evaluation of a decision statement, such as an if-statement. The statistics pertaining to branch coverage, as displayed in table 5.6 and graphically represented in figure 5.3, offer a similar perspective to the line coverage findings.

Test Run	Branches Total	Branches Covered	Branches Covered (%)
cangen 5	106	13	0.13
cangen 15	106	19	0.18
cangen 30	106	19	0.18
cangen 60	106	19	0.18
none 5	106	41	0.39
none 15	106	95	0.90
none 30	106	91	0.86
none 60	106	96	0.91
balanced 5	106	55	0.52
balanced 15	106	90	0.85
balanced 30	106	97	0.92
balanced 60	106	97	0.92
aggressive 5	106	96	0.91
aggressive 15	106	89	0.84
aggressive 30	106	94	0.89
aggressive 60	106	97	0.92

Table 5.6: Branch coverage within each test run of the stateful target. Test runs are executed for 5, 15, 30 and 60 minutes by the tool cangen and the prototype fuzzer with snapshot scheduling of none, balanced and aggressive. For all runs, three fuzzing instances run simultaneously

During fuzzing operations with the cangen tool, branch coverage remains below 18%, significantly lower than what is achieved when employing the prototype fuzzer developed within this thesis.

When employing the fuzzer's none scheduling option, branch coverage reaches only 39% after 5 minutes. However, this coverage improves notably when the fuzzer runs for an extended period. But, it does not match the performance achieved when using the balanced or aggressive options.

The balanced snapshot option initiates with a 52% branch coverage in the 5 minute test run and subsequently reaches a peak of 92% during the 30 and 60 minute runs.

The aggressive snapshot scheduling option in the fuzzer maintains consistently high branch coverage in all test runs, with the lowest being 84% in the 15 minute test run and the highest at 92% in the 60 minute test run.

Within the experiments executed in this thesis, it can be concluded that the choice of snapshot scheduling options from the prototype fuzzer has a more significant impact

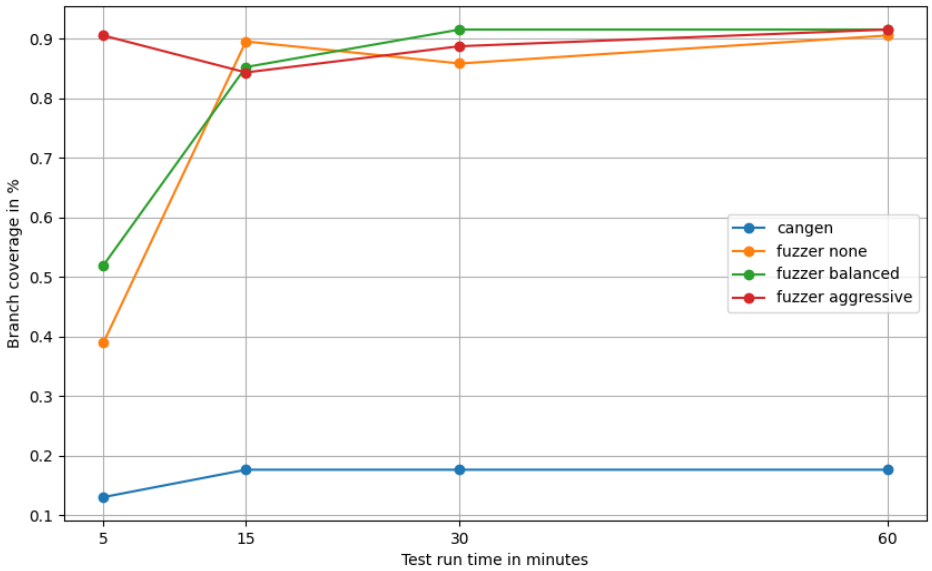

Figure 5.3: Branch coverage comparison of each test run of the stateful target..

when the testing period is shorter. This is likely because, with intermediate snapshots, the fuzzer can explore new program paths more rapidly. During the experiments the performance differences among different snapshot options become less pronounced when running for 15 minutes or longer. This could be attributed to the target program's relative lack of complexity, which limits the advantage of different snapshot scheduling options in the long run.

To reproduce the test cases, first run *all_canvehicle.sh* with the respective snapshot configuration set, then execute *canvehicle_rust_fuzzer_debug.sh*. To get coverage information, build the target program with

make CC="gcc" CFLAGS="-fprofile-arcs -ftest-coverage –coverage -g -O0"

Use *can-send-data.py* in canfd mode and provide the path to the fuzzing input created by running the fuzzer in debug mode. This is normally

/tmp/workdir/corpus/crash_reproducible/cnt_<X>.py or

/tmp/workdir/corpus/normal_reproducible/cnt_<X>.py

After that, use the tool *gcovr* [19] for displaying the coverage information.

[19] `https://gcovr.com/en/stable/`

CHAPTER 6

Future Work and Research Directions

Throughout this thesis, a prototype fuzzer tailored to the SocketCAN interface is created. This can be seen as a first practical step within the research domain of fuzzing real-world CAN implementations, as well as extending the scope to fuzzing embedded protocols.

This chapter starts with section 6.1 describing the requirements for extending the fuzzer's capability to encompass CAN protocol implementations that are more commonly employed in practical scenarios. Section 6.2 uses this knowledge to discuss the process of transforming the fuzzer created in this thesis to work for embedded protocols in general.

6.1 SocketCAN to CAN

Unlike other fuzzing tools outlined in section 2.3, when employing the fuzzer developed in this thesis, the target programs operate within a virtual machine created by QEMU. To preserve the advantages of snapshot-based fuzzing, maintaining this approach is preferable.

Currently, the fuzzer can only manage targets that employ standard C sockets, which isn't common for most applications utilising the CAN protocol. A necessary adjustment is to make communication between the fuzzer and the target program work for different technologies. Additionally, there might be challenges in detecting crashes during fuzzing the target program. By now, the fuzzer identifies crashes when the target program process terminates abnormally. This functionality must be maintained or the monitoring logic of the fuzzer adapted.

When extending the fuzzer's applicability to CAN protocol implementations beyond SocketCAN, the following new questions emerge:

1. Can the new targets deployed with a CAN protocol implementation be executed within the fuzzer's provided virtual machine environment?
2. Does the target program need specific CAN support inside the virtual machine?
3. Is it possible to inject fuzzing data into the target in a similar way as with SocketCAN?
4. Are crashes of the target program still discovered by the fuzzer?
5. Can a crash be reproduced without a bare-metal setup as depicted in figure 2.2?

If questions 1-4 are answered with yes, the fuzzing setup shown in figure 2.2 can be transformed to the one outlined in figure 6.1. The CAN Board and CAN Bus components are not needed anymore because there is no real communication through a CAN bus happening during fuzzing. To reproduce a found crash, a configuration similar to the one depicted in figure 2.2 is required, or at the very least, the capability to emulate a CAN interface besides SocketCAN on the host system must be present. The possibility of it must still be evaluated.

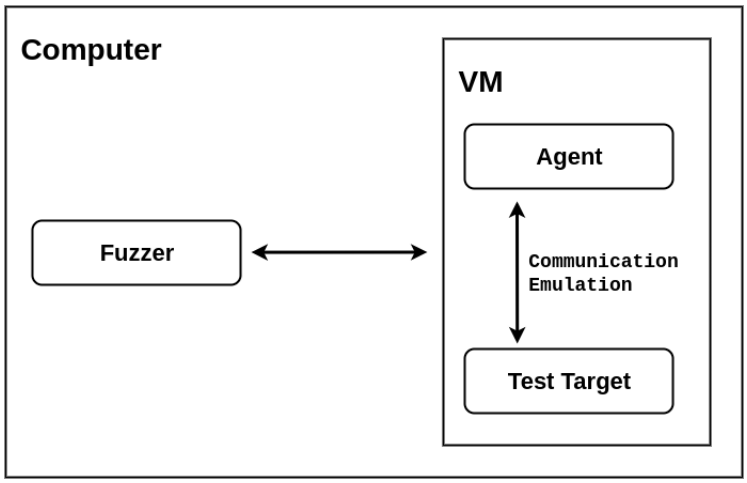

Figure 6.1: Thesis' fuzzer CAN fuzz setup idea.

6.2 Fuzzing Embedded Protocols

The outline provided for transitioning the fuzzer from SocketCAN to CAN effectively serves as a foundational framework for discussing embedded protocols at large. In this part of the thesis, the questions raised in the preceding section are extrapolated or directly applied to encompass embedded protocols in a broader sense. Consequently, this generates five key questions to the broader integration of the existing fuzzer for diverse embedded protocols:

1. How can the targets be executed within the fuzzer's provided virtual machine environment?
2. Is there the need for specific virtual machine configuration/support regarding the utilised embedded protocol?
3. How does the communication between fuzzer and target work?
4. Are crashes of the target program still discovered by the fuzzer?
5. How can a crash be reproduced?

The remaining part of this section explores these questions in more detail.

How can the targets be executed within the fuzzer's provided virtual machine environment?

In the current developmental stage of the fuzzer, the target programs are executed within a virtual machine alongside the *Agent* component. This virtual machine adopts a Linux-based operating system and operates on a 64-bit x86 architecture emulated through QEMU. A custom version of QEMU is used, called QEMU-Nyx [1]. This modified version of QEMU is tailored for the Nyx fuzzing framework, which is also employed in the prototype for this thesis. While the adapted QEMU version is obligatory for the correct functioning of the fuzzer, there remains the flexibility to customize or expand it to suit specific needs. The specific virtual hardware configuration involves a customized kAFL64 architecture, with a corresponding kAFL64-Hypervisor CPU model. A notable prerequisite for the target program is its compatibility with the afl-clang-fast or afl-clang-fast++ compiler [2]. This should not be a big problem as a vast amount of firmwares are written in a language of the C family.

Firmware are in general closely coupled to the hardware they are running on. A illustration of this is the Klipper 3D printer firmware [3], which, when equipped with CAN support, can solely be employed on microcontrollers like STM32 [4], SAME5x [5], and rp2040 [6]. Embedded software is designed to also run on a particular device. Compared to (embedded) firmware it controls higher-level functions of the device. In practice, the distinction between software and firmware can sometimes be blurry, thus for the remainder of this section, *firmware* will encompass both. Given the hardware interdependency, one of the main challenges is that publicly available firmware may not be compiled and executed within the virtual machine currently provided by the fuzzer.

[1] https://github.com/nyx-fuzz/QEMU-Nyx
[2] https://manpages.ubuntu.com/manpages/xenial/man1/afl-clang-fast.1.html
[3] https://www.klipper3d.org/
[4] https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html
[5] https://microchipdeveloper.com/32bit:same5
[6] https://www.raspberrypi.com/documentation/microcontrollers/rp2040.html

In the absence of adjustments to the existing virtual machine configuration, the task of identifying fitting firmware for fuzzing becomes challenging. This extends to both implementations of the CAN protocol and embedded protocols at large. As the current virtual machine setup demonstrates limitations in accommodating a diverse range of firmware variations, a need arises to refine the setup to a wider scope of firmware types. Operating systems based on the Linux kernel are widely used in embedded systems such as consumer electronics, industrial automation and spacecraft flight software. This means by having a Linux OS installed in the virtual machine of the fuzzer, it should be still possible to find firmware which can run inside this operating system. A bigger problem could be the used machine type (kAFL64) and CPU model (kAFL64-Hypervisor) in the virtual machine of the fuzzer. An assessment is needed to determine the viability of the current configuration, or if modifications are required to avoid disrupting the fuzzer's operations. Should the QEMU configuration of the fuzzer be reconfigured to emulate an architecture suitable for a specific firmware, a comprehensive guide outlining the modification process would be valuable. Given that various firmwares necessitate distinct configurations, a comprehensive understanding of adapting the fuzzer and the ability to implement changes would significantly enhance the fuzzer's potential to test a wider variety of firmwares.

Is there the need for specific virtual machine configuration/support regarding the used protocol?

Enabling fuzz testing for a target program utilising a communication protocol implementation other than TCP sockets and SocketCAN shouldn't pose significant challenges in regard of the virtual machine setup. This is the case because the fuzzer intercepts all receive and write functionalities during the fuzzing process, ensuring that no actual communication occurs. While there are other factors to be addressed for seamless operation of the interceptions, these aspects will be discussed in the subsequent question. In scenarios where more complex programs involve multiple communication pathways, the ideal scenario is that all these pathways are handled by the fuzzer's function hooks. However, practicality might dictate that not all communication functions are feasibly hooked. In such instances, it becomes inevitable that the unhooked communication functions can execute accurately.

The process of integrating a new target into the fuzzer can result in substantial time savings by not implementing virtualisation support for a new communication protocol. But, as in the case of the SocketCAN development it turned out to be very useful to have CAN support added to the QEMU virtual machine, and saved time in the long run. The decision if it is necessary to to have the support is best taken by the developer who adapts the fuzzer.

If the decision is made to incorporate virtualisation support for a new protocol, several factors should be taken into account. QEMU supports an extensive range of protocols that are applicable, even in the realm of embedded systems. Reflecting on the experience of developing the SocketCAN extension for the fuzzer, it turned out that having accessible

transmitted data on the host system was beneficial, although it did require some effort to achieve. Additionally, toggling specific functionalities related to SocketCAN was found to be valuable. This practice could also be applied when developing extensions for new protocols. Enabling and disabling functionalities can be best facilitated by introducing an environment variable, similar to CAN_MODE, with an analogous approach for the new protocol's support.

A guide similar to the one provided for the CAN protocol in section 4.3 can prove valuable when dealing with the integration of new protocols. Different protocols differ in their implementation but it is still useful to have access to such a guide because it can help during the debugging process in the event of any issues and helps in adding support for additional communication protocols. This approach not only promotes consistency across various protocol implementations but also facilitates the expansion of the fuzzer's compatibility with additional communication protocols.

How does the communication between fuzzer and target work?

The socket emulation is one of the pivotal aspects of the fuzzer crafted within the scope of this thesis. In section 3.6 the technology behind it is explained briefly. An LD_PRELOAD interceptor is used to hook and transform the functionalities of the functions employed by the target program for data reception and transmission. Even functions pertaining to the management of socket connections are effectively overridden, thereby enabling the establishment of a custom connection management framework within the fuzzer. While currently, this approach is confined to libc functions, its extensibility spans the realm of any library calls. Hence, this implies the possibility for extending the fuzzer's functionality to hook other communication related functions in diverse protocols.

In relation to SocketCAN, a new mode named CAN_MODE has been introduced. This addition serves a specific purpose: to tailor the behaviour of certain socket functions based on the socket type. While this mode's creation is rooted in the need of having hooks behave differently for the same function, the concept of introducing new modes could be beneficial for accommodating potential variations in other protocols as well.

When venturing into fuzzing a new protocol-based target, a new target input specification has to be created to make the fuzzing process as efficient as possible. The description of the input specification in section 4.8 can be used as a guideline. This tailored specification aims to optimize the fuzzing process by incorporating seed files and implementing their parsing to extract payload data. This extracted data is subsequently employed by the mutation engine to craft the fuzz test inputs. While the process of obtaining seed files might vary depending on the target, the approach employed so far involves running the target locally and recording received data using tools like Wireshark. Although this method might prove more intricate for other targets, manual provision of suitable seeds remains a viable alternative. Extracting data from these seeds should follow a process similar to that used for the SocketCAN targets, with only minor differences in naming conventions expected.

Most time spend during this adaptation process will center around adjusting the connection management and function hooks. For instance, in the case of SocketCAN, the connection management was expanded to accommodate CAN socket connections. The structure of the code implies that for each new protocol introduced, the likelihood of having to modify or extend the connection management system is quite high, prompting a consideration of potential redesigns.

Function hooks responsible for injecting data into the target, must provide the input data in the same type definition as defined in the target program, a task often best accomplished by replicating the type definition and populating values as needed. These values can often be draw from other function hooks.

Are crashes of the target program still discovered by the fuzzer?

The challenge of detecting crashes in embedded systems is a well-recognized concern within the domain of fuzzing. This challenge becomes particularly pronounced due to the unique characteristics of embedded operating environments. This issue has been explored, as highlighted in the discussion presented in section 2.1. The intricacy arises from the fact that embedded operating systems often lack the comprehensive fault generation mechanisms commonly found in conventional systems. Moreover, the absence of conventional output devices like monitors further complicates the task of identifying crashes in such environments. The architecture of the developed fuzzer, runs the target program within a Linux operating system, thus the fuzzer can effectively detect abnormal process terminations and closely monitor any occurrences of crashes. This inherent capability is derived from the Linux environment's intrinsic mechanisms for recognizing anomalous program behaviour. A wide spectrum of available embedded system programs, often functioning within Linux operating systems, ensures that a diverse array of target programs can be effectively monitored for crashes. Consequently, the current fuzzer setup stands to benefit from the existing robust crash detection mechanisms without necessitating any fundamental changes in this regard.

If the virtual machine setup transition towards a less powerful operating system environment, problems regarding the crash detection will arise. As the fuzzer might not receive sufficient information to effectively evaluate whether a crash has occurred, it may not be able to report found crashes to the user. This situation arises due to the altered contextual information provided to the fuzzer within the new environment. Addressing this challenge requires exploration of alternative strategies for crash detection. In this context, there are broadly known solutions to ensure effective crash detection even in constrained operating environments. For example liveness checks make it possible to periodically probe the state of the device, memory checking tools can detect security violations, and debug ports can help to gain insight into the device. However, the integration of these alternative methods into the existing fuzzer infrastructure necessitates a fundamental shift in the core functionality of the tool. Specialising the fuzzer to accommodate embedded systems while potentially relinquishing support for classical network targets. If such changes in the fuzzer are made, the methodology outlined in chapter 5 can serve as a useful guide

to test the new functionalities. A custom test target featuring a distinct vulnerability could be employed to assess the fuzzer's new crash detection mechanisms.

The behaviour of crash detection is not closely tied to the specific communication protocol employed. Rather, it depends on the configuration of the virtual machine and the manner in which the target application addresses anomalous behaviour. A solution devised for a particular setup and application behaviour should have the potential to be applicable across various protocols that share the same underlying setup and program behaviour characteristics.

How can a crash be reproduced?

A comprehensive guide outlining the procedure for reproducing a crash in the fuzzer's present development state is expounded upon in section 4.10. To succinctly summarize, once the fuzzer identifies a crash, it must be executed in debug mode to translate the injected input into a usable format. Subsequently, the target program is run on the local system, and a Python script is employed to transmit the crash input to the target, ultimately leading to the occurrence of a crash.

When we focus solely on target programs that can be executed within a Linux virtual machine, the existing crash reproduction workflow remains largely unaffected. The transition to this new context primarily involves adjusting the previously mentioned Python script, tailoring it to align with the characteristics of the new target. Alternatively, for those inclined, the process of sending a specific input to the target can be executed manually, thereby eliminating the necessity for the Python script. It's worth noting that some scenarios might involve target programs crashing only after a sequence of inputs is received. In such cases, it proves more convenient to create an additional script rather than manually executing the steps, especially when dealing with multiple crashes in the same program. This systematic approach ensures a consistent and efficient procedure for crash reproduction, irrespective of the specific target and its intricacies.

In the event of a change in the virtual machine setup, additional factors need consideration. The main question is about the setup in which the target program now operates. It's conceivable that the target program may no longer be feasible to run directly on the host system. If such a scenario arises, new hardware has to be obtained, or virtualisation is used. Using virtualisation the same configuration can be applied as used during the fuzzing process. Subsequently, the target program would be executed within this virtual environment, and the target input could be dispatched either manually or via a script. This particular step in the process is unlikely to pose substantial challenges, considering that the virtual machine's configuration has already been established. As previously mentioned, QEMU offers extensive hardware emulation capabilities, encompassing a wide array of hardware configurations. This inherent flexibility should facilitate the fuzzing of target programs with diverse hardware or other requirements, providing the necessary adaptability to accommodate potential variations in the virtual machine environment.

CHAPTER 7

Conclusion

The integration of embedded systems within a broad range of industries necessitates a resilient strategy to mitigate potential vulnerabilities. The ramifications of any vulnerabilities or disruptions can be dire. Unfortunately, the security of these systems often leaves much to be desired. Identifying and rectifying vulnerabilities prior to their exploitation by malicious entities holds the potential to significantly bolster the security of these systems.

In this context, fuzzing emerges as a notably effective approach to achieve this objective. In order to foster security research on embedded system this thesis expands the reach and relevance of contemporary fuzzing techniques in the realm of systems employing protocols specific to embedded systems.

This thesis expands the capabilities of the open-source fuzzer Nyx-Net to encompass programs utilising the SocketCAN protocol for communication. The enhanced fuzzer operates on the latest version of Ubuntu, which, as of the time of writing, is Ubuntu 22.04 LTS. It offers user-friendly scripts to streamline the utilisation of the fuzzer. Additionally, it integrates support for the CAN protocol within the virtual machine where the target program executes. Furthermore, the development entails crafting an individualised input specification tailored for targets that receive CAN-specific data. The process of injecting this custom input into the targets is fitted to SocketCAN communication. The newly implemented functionalities are evaluated and tested across various target programs.

In Chapter 6, forthcoming challenges in the expansion of the fuzzer's capabilities are discussed, accompanied by solutions and ideas. Five questions are raised to discuss topics regarding the fuzzer's virtual machine setup, the intricacies of communication with the target and crash management. The ideas provided lay the path for using the fuzzer in relation to other embedded protocols.

Chapter 1 introduces specific research questions concerning the fuzzing of embedded protocols. The remainder of this chapter aims to provide answers to these inquiries.

RQ1: What are the key requirements and considerations for adapting the Nyx-Net fuzzer to effectively test and fuzz a program using the protocol implementation SocketCAN, and how can these challenges be addressed?

To facilitate further research, the Nyx-Net fuzzer is expanded to encompass target programs employing the SocketCAN protocol. Since Nyx-Net is inherently designed for network applications rather than embedded systems, various modifications are required across its components. For a comprehensive understanding of these adaptations and extensions, the Background chapter 3 delves into the technologies utilised within Nyx-Net and how they are integrated. This chapter also furnishes a detailed exposition of Nyx-Net, with a specific focus on the areas that underwent alterations.

In the subsequent chapter 4, the necessary fixes, adaptations, and new features for the development of a new fuzzer based on Nyx-Net are explained. The fuzzing setup initially encountered challenges with the Ubuntu 22.04 LTS operating system. Furthermore, the virtual machine configuration is augmented to include support for CAN, necessitating an update to the Linux kernel to enable certain features. One of the primary challenges revolved around establishing accurate communication between the target program and the fuzzer, primarily due to the distinct data structures employed by SocketCAN targets. The introduction of new functionalities such as CAN_MODE and CAN_SPEC_TEST_MODE allows for seamless fuzzing of SocketCAN targets without disrupting Nyx-Net's existing functionalities, and additionally aiding in the development of the new fuzzer.

RQ2: How can the modifications to Nyx-Net be implemented to preserve its original functionality while incorporating the new changes?

To have the fuzzer not only work for the SocketCAN protocol implementation, but also for the targets intended by the used foundation which is the fuzzer Nyx-Net, there should be a way to support both type of targets. This challenge is effectively addressed through the introduction of a new mode called CAN_MODE. By default, this mode remains disabled, ensuring that the fuzzer operates in a manner consistent with Nyx-Net. However, when this mode is activated, the fuzzer becomes capable of effectively fuzzing programs that rely on data received through SocketCAN.

At one hand, enabling CAN_MODE leads to adjustments in the virtual machine configuration wherein the target program executes. Additionally, it entails adaptations in the fuzzer's communication approach with the target program. For a comprehensive understanding of the implementation and technical aspects of this mode, refer to the same chapter as discussed in RQ1, namely chapter 4.

RQ3: How can the findings and insights gained from successful fuzzing of a target utilising SocketCAN be leveraged to advance fuzzing research on other CAN protocol implementations?

Given that there exist various implementations of the CAN protocol beyond SocketCAN, broadening the fuzzer's scope to encompass one of these alternatives can be seen as a natural next step. An ideal candidate for this expansion could be a program employed in automotive systems. However, it's important to note that such programs are often not publicly accessible. Therefore, successful integration would likely necessitate collaboration with the respective manufacturers.

In Section 2.3, an overview of existing tools for fuzzing applications employing the CAN protocol is provided. Notably, all of these tools have relied on actual hardware within their fuzzing setups.

In Section 6.1, an exploration of challenges and potential solutions related to adapting the existing fuzzer for other CAN protocol implementations is presented. This section outlines five pivotal questions that address the intricacies when extending the fuzzer's capabilities to encompass new CAN protocol variants. These inquiries center around aspects such as the virtual machine setup, bidirectional communication with the target program, and the crucial matters of crash detection and reproducibility.

RQ4: How can the knowledge acquired from successful fuzzing of a target using SocketCAN be applied to advance research on embedded protocols in general?

The insights derived from the experience of fuzzing SocketCAN applications and the study of CAN targets can be leveraged to derive the potential use of the developed fuzzer for various other embedded protocols. Section 6.2 takes the lessons learned from fuzzing SocketCAN to encompass other CAN protocols, and broadens the approach to cater to a more generalized scope. Five key questions are raised, revolving around the same topics as the questions raised for RQ3, namely virtualisation, target communication and crash processing.

List of Figures

List of Tables

Acronyms

AFL American fuzzy lop. 7, 11, 35, 55

API application programming interface. 3, 19, 21, 25, 26

AST Abstract Syntax Tree. 22, 23

CAN Controller Area Network. ix, 1, 3–5, 7, 13–15, 17–22, 31, 32, 36–38, 41, 44–49, 54–58, 61–65, 67, 75–80, 83–85, 87

CAN FD Controller Area Network Flexible Data-Rate. 13, 19, 55–57, 62–65, 67

CPU Central Processing Unit. 26

CUDA Compute Unified Device Architecture. 22

DDoS distributed denial-of-service. 1

ECU Electronic Control Unit. 17

IDE Integrated Development Environment. 22

IoT Internet of Things. 1, 11, 12

KVM Kernel-based Virtual Machine. 22, 24, 26, 34

OS Operating System. 10, 16, 17, 20, 23, 24, 26, 78

QEMU Quick Emulator. 4, 5, 15, 21, 22, 24, 26, 29, 36, 39, 75, 77, 78, 81

TCP/IP Transmission Control Protocol/Internet Protocol. 19

VM Virtual Machine. 15, 20, 22, 24–26, 30, 36, 37, 39, 40

Bibliography

[1] Bind documentation. https://man7.org/linux/man-pages/man2/bind.2.html. [Online; accessed 13-July-2023].

[2] C++ testing framework for writing and executing fuzz tests. https://github.com/google/fuzztest. [Online; accessed 30-July-2023].

[3] Can injection: keyless car theft. https://kentindell.github.io/2023/04/03/can-injection/. [Online; accessed 10-July-2023].

[4] Cansend documentation. https://manpages.debian.org/testing/can-utils/cansend.1.en.html. [Online; accessed 17-July-2023].

[5] Cisco secure development lifecycle. https://www.cisco.com/c/en/us/about/trust-center/technology-built-in-security.html#~processes. [Online; accessed 30-July-2023].

[6] Connect documentation. https://man7.org/linux/man-pages/man2/connect.2.html. [Online; accessed 13-July-2023].

[7] Figure of standard can frame. https://www.csselectronics.com/pages/can-bus-simple-intro-tutorial. [Online; accessed 19-July-2023].

[8] Google oss-fuzz. https://google.github.io/clusterfuzz/. [Online; accessed 30-July-2023].

[9] Google oss-fuzz. https://github.com/google/oss-fuzz. [Online; accessed 30-July-2023].

[10] Llvm project. libfuzzer. https://llvm.org/docs/LibFuzzer.html. [Online; accessed 30-July-2023].

[11] Microsoft security development lifecycle. https://www.microsoft.com/en-us/securityengineering/sdl/practices. [Online; accessed 30-July-2023].

[12] The new jersey cybersecurity and communications integration cell (njccic). [n.d.]. mirai botnet. https://www.cyber.nj.gov/threat-profiles/botnet-variants/mirai-botnet. [Online; accessed 10-July-2023].

[13] Nyx-packer documentation. https://github.com/nyx-fuzz/packer/tree/afa19b6b3c62ad043408a98b29f1e716a88951ce. [Online; accessed 16-July-2023].

[14] Qemu can support documentation. https://www.qemu.org/docs/master/system/devices/can.html#. [Online; accessed 13-July-2023].

[15] Socket documentation. https://man7.org/linux/man-pages/man2/socket.2.html. [Online; accessed 13-July-2023].

[16] Socketcan documentation. https://www.kernel.org/doc/html/latest/networking/can.html. [Online; accessed 11-July-2023].

[17] Bernhard K Aichernig, Edi Muškardin, and Andrea Pferscher. Learning-based fuzzing of iot message brokers. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 47–58. IEEE, 2021.

[18] Anastasios Andronidis and Cristian Cadar. Snapfuzz: high-throughput fuzzing of network applications. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 340–351, 2022.

[19] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[20] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. Fudge: fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 975–985, 2019.

[21] Jake L Beavers, Michael Faulks, and Jims Marchang. Hacking nhs pacemakers: a feasibility study. In *2019 IEEE 12th International Conference on Global Security, Safety and Sustainability (ICGS3)*, pages 206–212. IEEE, 2019.

[22] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46. Califor-nia, USA, 2005.

[23] Marcel Böhme and Brandon Falk. Fuzzing: On the exponential cost of vulnerability discovery. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 713–724, 2020.

[24] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 122–131. IEEE, 2013.

[25] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS*, volume 1, pages 1–1, 2016.

[26] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS*, 2018.

[27] Libo Chen, Quanpu Cai, Zhenbang Ma, Yanhao Wang, Hong Hu, Minghang Shen, Yue Liu, Shanqing Guo, Haixin Duan, Kaida Jiang, et al. Sfuzz: Slice-based fuzzing for real-time operating systems. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 485–498, 2022.

[28] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. Time-travel testing of android apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 481–492, 2020.

[29] Xuechao Du, Andong Chen, Boyuan He, Hao Chen, Fan Zhang, and Yan Chen. Afliot: Fuzzing on linux-based iot device with binary-level instrumentation. *Computers & Security*, 122:102889, 2022.

[30] Maialen Eceiza, Jose Luis Flores, and Mikel Iturbe. Fuzzing the internet of things: A review on the techniques and challenges for efficient vulnerability discovery in embedded systems. *IEEE Internet of Things Journal*, 8(13):10390–10411, 2021.

[31] Max Eisele, Daniel Ebert, Christopher Huth, and Andreas Zeller. Fuzzing embedded systems using debug interfaces. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023).*, 2023.

[32] Max Eisele, Marcello Maugeri, Rachna Shriwas, Christopher Huth, and Giampaolo Bella. Embedded fuzzing: a review of challenges, tools, and solutions. *Cybersecurity*, 5(1):1–18, 2022.

[33] Guy Farrelly, Michael Chesser, and Damith C Ranasinghe. Ember-io: Effective firmware fuzzing with model-free memory mapped io. *arXiv preprint arXiv:2301.06689*, 2023.

[34] 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.

[35] Daniel S Fowler, Jeremy Bryans, Siraj Ahmed Shaikh, and Paul Wooderson. Fuzz testing for automotive cyber-security. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 239–246. IEEE, 2018.

[36] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. Dlfuzz: Differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 739–743, 2018.

[37] Steve Heath. *Embedded systems design*. Elsevier, 2002.

[38] Thomas A Henzinger and Joseph Sifakis. The discipline of embedded systems design. *Computer*, 40(10):32–40, 2007.

[39] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, 2012.

[40] Arne Holst. Number of iot connected devices worldwide 2019–2030. *Statistica*, 2021.

[41] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. Winnie: Fuzzing windows applications with harness synthesis and fast cloning. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021)*, 2021.

[42] Hyeryun Lee, Kyunghee Choi, Kihyun Chung, Jaein Kim, and Kangbin Yim. Fuzzing can packets into automobiles. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*, pages 817–821. IEEE, 2015.

[43] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.

[44] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019.

[45] Barton P Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[46] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS*, 2018.

[47] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[48] Ryosuke Nishimura, Ryo Kurachi, Kazumasa Ito, Takashi Miyasaka, Masaki Yamamoto, and Miwako Mishima. Implementation of the can-fd protocol in the fuzzing tool bestorm. In *2016 IEEE International Conference on Vehicular Electronics and Safety (ICVES)*, pages 1–6. IEEE, 2016.

[49] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18, 2017.

[50] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: A greybox fuzzer for network protocols. In *Proceedings of the 13rd IEEE International Conference on Software Testing, Verification and Validation : Testing Tools Track*, 2020.

[51] Pavel Pisa, Jin Yang, and Michal Sojka. Qemu can controller emulation with connection to a host system can bus. In *17th Real Time Linux Workshop, Graz, Austria*, 2015.

[52] Mengfei Ren, Xiaolei Ren, Huadong Feng, Jiang Ming, and Yu Lei. Security analysis of zigbee protocol implementation via device-agnostic fuzzing. *Digital Threats: Research and Practice*, 2022.

[53] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using precise {MMIO} modeling for effective firmware fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1239–1256, 2022.

[54] Tobias Scharnowski, Felix Buchmann, Simon Wörner, and Thorsten Holz. A case study on fuzzing satellite firmware.

[55] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2597–2614, 2021.

[56] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *USENIX Security Symposium*, 2017.

[57] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-net: Network fuzzing with incremental snapshots. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, 2022.

[58] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *2019 Network and Distributed Systems Security Symposium (NDSS)*, pages 1–15. Internet Society, 2019.

[59] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent Byunghoon Kang, Jean-Pierre Seifert, and Michael Franz. Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2541–2557, 2020.

[60] Ed Sutter. *Embedded systems firmware demystified*. CMP books, 2002.

[61] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *Computer Security–ESORICS 2016: 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I 21*, pages 581–601. Springer, 2016.

[62] Joobeom Yun, Fayozbek Rustamov, Juhwan Kim, and Youngjoo Shin. Fuzzing of embedded systems: A survey. *ACM Computing Surveys (CSUR)*, 2022.

[63] Michał Zalewski. american fuzzy lop. `http://lcamtuf.coredump.cx/afl/`. [Online; accessed 13-June-2023].

[64] 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.

[65] Haichun Zhang, Kelin Huang, Jie Wang, and Zhenglin Liu. Can-ft: A fuzz testing method for automotive controller area network bus. In *2021 International Conference on Computer Information Science and Artificial Intelligence (CISAI)*, pages 225–231. IEEE, 2021.

[66] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. {FIRM-AFL}:{High-Throughput} greybox fuzzing of {IoT} firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114, 2019.

[67] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.