Fuzzing Android Automotive's CAN interface

Mihai Macarie
Master Thesis

Semantics, Cybersecurity & Services (SCS)
Faculty of Electrical Engineering, Mathematics & Computer Science
University of Twente
PO Box 217, 7500 AE, Enschede, Netherlands

Abstract. Our research aims to evaluate the cybersecurity of the Controller Area Network (CAN) [1] interface in Android Automotive using fuzzing techniques. The growing dependency of the automotive industry on cyber-physical systems exposes vehicles to new cyber risks and threats [2]. In addition, vehicles nowadays have external connections such as Bluetooth, WiFi, and mobile networks. Previous research has uncovered numerous security issues in these systems, including unencrypted protocols and privacy concerns [3, 4, 5]. In March 2017, Google introduced Android Automotive OS, an in-vehicle infotainment (IVI) operating system (OS). This operating system interacts with climate control and digital instrument clusters [6]. Thus, cyberattacks targeted at this OS endanger vehicle safety and, as a result, in some cases, also human lives. Polestar and Volvo use Android Automotive OS, and more manufacturers plan to use it [7, 8, 9]. Researchers have started investigating the security aspects of Android Automotive, but further research is necessary. In addition, there is no research on fuzzing specific components of Android Automotive [5, 10]. Fuzzing might identify software bugs that other testing techniques might not find. We perform fuzzing experiments on the CAN interface of Android Automotive, one of the most critical buses used in modern vehicles. We use libFuzzer and AFL for our experiments because of their integration into Android Open Source Project (AOSP) and their features. We perform experiments on AOSP emulators and car manufacturer emulators. We have noticed that AFL found several crashes during our experiments, while libFuzzer found nothing. We have also developed a modified harness that achieves higher code coverage. Furthermore, we observe that the version of the Android Automotive emulator used affects the code coverage. Finally, we have some contributions to the AFL++ fork in the repositories of AOSP.

Keywords: Cybersecurity · Fuzzing · Android Automotive · AFL · libFuzzer · CAN · Automotive

Committee members:

Chair:	dr.ir. Andrea Continella
UT member:	prof.dr.ir Roland van Rijskwijk-Deij
1st TNO supervisor:	Bart Marinissen, MSc.
2nd TNO supervisor:	Gerben Broenink, MSc.

Table of Contents

List of Figures

List of Tables

1 Introduction

The growing dependency of the automotive industry on cyber-physical systems exposes vehicles to new cyber risks and threats. Some examples include systems that detect road traffic signs, traffic lights, pedestrians, and vehicle recognition. In this category, we also add systems that detect blind spots and systems that detect lane departure and correct it, if necessary. In addition, vehicles nowadays have external connections such as Bluetooth, WiFi, and mobile networks. Previous research has uncovered numerous security issues in these systems, including unencrypted protocols and privacy concerns [3, 4, 5]. Besides academia, standards development organizations have created international standards such as ISO 26262 [12], UNECE R155 [13], and UNECE R156 [14], which are related to general cybersecurity in automotive systems, but also to automotive software. Even if there are cybersecurity standards that automotive industry stakeholders must follow, security issues can still be present. Cybersecurity standardization is challenging due to evolving threat landscape and potentially missing specific cases and scopes.

In March 2017, Google introduced Android Automotive, an Android OS variant for in-vehicle use. Its adoption has been growing among car manufacturers such as Volvo and Polestar. Therefore, more research on the security aspects of this OS is needed. Android Automotive interacts with critical vehicle protocols and systems like the Controller Area Network (CAN), and any potential vulnerability could seriously threaten human lives. By accessing the CAN bus, an attacker can sniff or inject malicious packets related to different systems in a vehicle, such as air conditioners and steering control [9, 10, 15].

We research the cybersecurity aspects of Android Automotive using fuzzing techniques and the fuzzing performance in such an environment. Fuzzing is a technique for detecting software bugs and vulnerabilities using random or semi-random input data and identifying potential crashes, unexpected behaviour, or information leaks using a fuzzer tool. We use fuzzing because it helps to uncover security vulnerabilities and overlooked software bugs during traditional testing approaches, representing a time and cost-efficient automated process. For a successful fuzzing process (i.e., discovering crashes and achieving the highest code coverage possible), we need the right fuzzing tools, good harnesses, and valuable initial inputs (input seeds). In fuzzing, the number of crashes detected and the code coverage achieved are essential because these metrics reveal how many potential security issues exist in the target and how the fuzzer reaches the target. Using fuzzing, we investigate whether the Android Automotive OS is reliable and safe to use, and we report any issues found to the appropriate stakeholders to be taken care of [16, 17, 18].

Android Automotive has many input interfaces of interest regarding fuzzing. However, we prioritize the interfaces we fuzz. We continue our investigation into the Controller Area Network (CAN) interface by assessing its risk of security vulnerabilities and applicability to automotive systems. Previous academic re-

search has highlighted the importance of securing this interface, making it a candidate for fuzzing [5, 19].

Our methodology relies on white-box fuzzing, which benefits from the target's source code accessibility. Firstly, we compile the target's source code with code instrumentation and integrate it into the build configuration of the environment where fuzzing occurs. Secondly, we prepare the initial input seeds to kick-start the fuzzing process. Thirdly, fuzzing involves continuous analysis of target behaviour using input seeds, observing code coverage and system abnormalities, and generating new input seeds through mutation and evolutionary algorithms. Lastly, upon completing the fuzzing process, we review the findings.

Our experiments include comparisons between the existing CAN hardware interface fuzzing target in AOSP and our modified version that aims to improve code coverage and crash detection. We also investigate the role of initial data used in fuzzing. Finally, we compare the Android Automotive emulator built directly from the AOSP source and some of the emulators provided by General Motors (GM) and Honda.

The experiments reveal that AFL++ (AFL for simplicity) detects more crashes than libFuzzer, which has not detected any. AFL also achieves a higher coverage than libFuzzer in most cases. Furthermore, our modified version of the harness usually achieves higher coverage. Finally, we conclude that the Android version affects both harnesses' coverage.

We have contributed to the AOSP project by adding building rules for two AFL binaries. AOSP maintains a modified version of the AFL repository for Android OS, excluding some original AFL binaries because the maintainers still need to integrate them or are unnecessary in the Android environment.

1.1 Research questions

Our main research question for this master thesis is:

Research question 1. *How performant is the fuzzing in testing the CAN interface, and how can this performance be improved in the Android Automotive environment?*

We split this main research question into three other sub-questions. Therefore we introduce the first sub-research question:

Sub-Research question 1.1. *How does the harness's complexity affect the source code's code coverage or the number of crashes found?*

When we answer this question, we see how a modified harness affects the fuzzing performance when we answer this sub-question. Adding more calls in harnesses increases the code coverage.

Sub-Research question 1.2. *How do different fuzzers affect the fuzzing process performance in Android Automotive?*

Answering this sub-question helps us to understand how the libFuzzer and AFL perform on the same harnesses. Each fuzzer engine has its algorithms for performing fuzzing, which affects the fuzzing performance.

Sub-Research question 1.3. *How do different Android Automotive emulators affect the source code's code coverage or the number of crashes found?*

By answering this sub-question, we want to analyze the fuzzing performance of the harnesses on different Android versions. Each Android version has its implementation particularities, which can affect the fuzzing performance.

The aforementioned research questions and sub-questions guide us in achieving our goal of using fuzzing the CAN interface in the Android Automotive environment.

1.2 Hypotheses

Our study tests the following hypotheses, which correspond to our research questions:

1. Increasing the harness's complexity increases the source code's code coverage or the number of crashes found. This hypothesis helps us answer Subquestion 1.1. Modifying a fuzzing harness by adding more calls to the interface under test might lead the fuzzer to undiscovered parts in the source code of the interface, so increasing the code coverage and the chance of detecting crashes.
2. libFuzzer detects fewer crashes than AFL. libFuzzer and AFL are coverage-based fuzzers that use genetic algorithms. LibFuzzer uses evolutionary algorithms and heuristics based on feedback. AFL uses mutation techniques for creating corpus files and relies heavily on coverage feedback. Thus the likelihood of achieving a higher corpus count and discovering crashes is increased in the case of AFL [20,21,22]. By verifying this hypothesis, we answer Sub-question 1.2.
3. The version of the Android Automotive emulator influences the source code's coverage, the number of crashes or the type of crashes. This hypothesis helps us answer Sub-question 1.3. Each Android Automotive emulator version has its changes regarding the source code of the packages, services, and kernel. Older versions are more likely to be more vulnerable than the current ones.

1.3 Research challenges

The challenges we come across during our research are:

1. Simulation of CAN protocol communication – the simulation in a virtualized environment of a physical vehicle or vehicle-specific hardware subsystem is challenging because of the complexity of real-life situations (e.g., multiple sensors communicating at the same time in a vehicle and various vehicles operating situations), multiple types of frames in CAN protocol, and replicating security issues in CAN protocol in an emulated environment.
2. Fuzzing of hardware interface implementations - for achieving good code coverage, thus increasing the chance of detecting software abnormalities, it is crucial to write performant fuzzing harnesses. Writting performant harnesses is challenging because Android Automotive supports various hardware devices and influences fuzzing performance (e.g., code coverage and fuzzing speed). Fuzzing hardware interface implementations is challenging because of the necessity of particular configurations, limited device resources, and limited documentation.
3. Interacting with the Android Automotive emulator for fuzzing purposes – requires a good understanding of how the emulator communicates with the host machine. Interacting with the emulator is challenging because we need to understand the emulator and communication between the emulator and the host machine, use code instrumentation, hardware limitations, and concurrency and timing issues.

2 Background

This section presents background knowledge about the most relevant notions and technologies useful for our master thesis.

2.1 Fuzzing

Fuzz testing, or fuzzing, is a technique for discovering software bugs and vulnerabilities. The technique sends random or semi-random input data to a software system to identify potential crashes, unexpected behaviour, or information leaks. The tool used for performing fuzzing is called a fuzzer [16]. We usually measure the performance of fuzzers in terms of code coverage. Code coverage represents a metric that counts the number of lines, basic blocks, and other similar features processed while the software is executed [17, 18].

A common concept in fuzzing is code instrumentation, which inserts special instructions in certain parts of the code to track the code coverage. For example, we instrument mathematical, logical, and array operations to track the input flow throughout program execution. Fuzzing is a search problem that seeks to have as much code coverage as possible and find the best inputs to cause the most significant software crashes.

A fuzzer may rely or not on the system under test structures, such as variables, logical flows, input/output formats, and data types for performance improvement

(e.g., code coverage and speed). As a result, we tell the difference between the following fuzzer types [23, 24, 25]:

1. Dumb fuzzer - does not consider the system's existing structure.
2. Smart fuzzer - the chance of finding a crash is increased by creating meaningful input test data, for instance, using knowledge of the system's existing structures.

We also categorize fuzzers into the following types based on how much prior knowledge we have about the system under test as follows [23, 24, 25]:

1. White Box Fuzzing - fuzzing a system having good knowledge of the structures in the system, including build environment and source code, with complete monitoring of the code paths that the fuzzer has reached at that point. By calling conditional branches along the way, symbolic execution restricts the inputs used in the fuzzing process. Usually, we use symbolic execution in a dynamic environment. To achieve as many possible execution paths, we repeatedly use symbolic execution while utilizing cutting-edge search techniques. [26].
2. Black Box Fuzzing - fuzzing a system without knowing the system's structures, especially not the source code, without monitoring the code paths the fuzzer has reached. This type of fuzzing is recommended for large, nondeterministic, slow systems or complex input data (e.g., a JPEG image). Some examples of black-box fuzzers include Radamsa [27], BFuzz [28], and ClusterFuzz [29, 30, 31].
3. Grey Box Fuzzing - a combination of white and black box fuzzing, having partial knowledge of the structures in the system, including parts of the source code, with partial monitoring of the code paths that the fuzzer has reached. American Fuzzing Loop (AFL) [32] is a well-known example of a grey-box fuzzer.

Based on how we generate the input data, we divide fuzzers into three categories [23, 25]:

1. Generation fuzzer - data is entirely randomized or slightly prepared from scratch. Generally, the fuzzer slices valid input data into multiple parts and fuzz each piece randomly. Radamsa [27] and Peach Fuzzer [33] are examples of fuzzers using generation techniques researchers discuss in academia.
2. Mutation fuzzer - valid input data (input seed) is provided to the fuzzer. The fuzzer uses this input seed to generate more testing data using alteration techniques such as bit flipping. Several fuzzer use mutation approaches, such as libFuzzer [22], AFL [32], and Peach Fuzzer [33].
3. Evolution fuzzer - input data is generated using principles of genetic programming. Whenever the fuzzer identifies an abnormality by executing a particular input, such as a system malfunction or a memory leak, it uses

that input to create more testing data. Some examples of such fuzzers are fuddly [34] and VUzzer [35].

Fuzzing helps find vulnerabilities and bugs such as memory buffer errors (e.g., buffer overflow), data validation issues (e.g., out-of-bounds array indexes), improper pointer usage (e.g., dereference of NULL pointer), numeric errors (e.g., an integer overflow), concurrency issues (e.g., improper synchronization) and bad coding practices (e.g., stack variable address return) [36, 37].

2.2 Android Automotive and Controller Area Network (CAN)

Android Automotive is an Android OS variant designed for in-vehicle infotainment (IVI) systems. However, it includes extra libraries and features designed explicitly for IVI systems. An example is the hardware abstraction layer (HAL), which helps connect and integrate the existing networks and buses in a vehicle: e.g., CAN, LIN, WiFI, Bluetooth, and Ethernet [9].

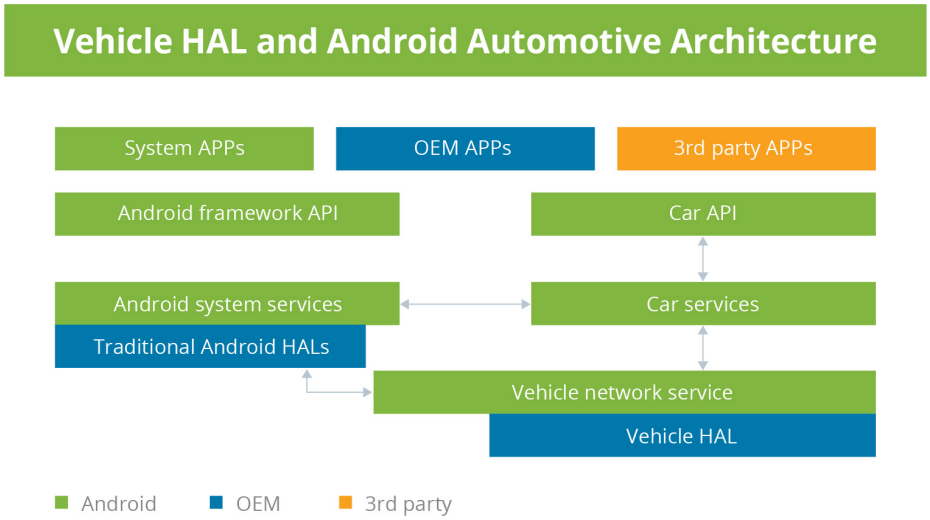

Fig. 1. Android Automotive architecture [7]

As illustrated in Figure 1, the Android Automotive ecosystem includes several stakeholders: Android, original equipment manufacturers (OEMs), and 3rd Parties. Android Automotive OS includes the following types of apps: systems apps (developed by Android), OEM apps (developed by OEMs), and third-party apps (developed by third parties) [38, 39].

These apps run on top of two Application Programming Interfaces (APIs): the Android Framework API and the Car API. The latter communicates with the

Car service, which communicates with the Android System Services. Moreover, the Vehicle HAL is a development interface for Android Automotive that "sets the properties OEMs can implement and contains property metadata". With the help of this HAL, the Traditional Android HALs and the Car Service communicate more easily.

The Controller Area Network (CAN) bus is a communication protocol widely used in the automotive industry, developed by Bosch in 1986. It uses a two-wire interface for communication inside a vehicle. The protocol operates on the OSI model's Data Link and Physical layers. CAN support efficient real-time communication, robust error detection, and handling capabilities, with data rates up to 1 Mbit/s. However, newer protocol versions, such as CAN-FD (Flexible Data-Rate), have higher data rates [1, 40, 41].

In Android Automotive, the CAN interface is part of the Vehicle HAL, part of the lowest level in the architecture shown in Figure 1. Within the AOSP source code, it is located in the `hardware/interfaces/automotive/can` directory. Android Code Search platform contains the source code of the interface[1]. The CAN interface communicates with and integrates various sensors and actuators present in the car, such as speed, engine parameters, air conditioner, and lightning.

2.3 Current guidelines in automotive cybersecurity

The automotive industry implements several standards to ensure safety, including cybersecurity. The three most relevant to our discussion are ISO 26262, UNECE R155, and UNECE R156 [2, 12, 13, 14, 42, 43, 44, 45, 46].

ISO 26262, also known as "Road vehicles – Functional safety", provides recommendations for the safety of the "electrical and electronic systems installed in serial production road vehicles". To ensure software security assurance, Automotive manufacturers can use certified commercial tools like Helix QAC [47], a static analysis tool for C and C++, and Mayhem [48], a software security testing tool using that also includes fuzzing.

UNECE R155, named "Cyber security and cyber security management system", is a set of regulations specially focused on the cybersecurity management aspect of automotive products. This standard emphasizes integrating cybersecurity measures into vehicle design and operation to ensure optimal security. However, the standard does not explicitly mention fuzzing as a way of testing software but encourages manufacturers to use tools for vulnerability detection.

UNECE R156, titled "Software update and software update management system", sets regulations for software updates in the automotive industry. These

[1] https://cs.android.com/android/platform/superproject/+/master:hardware/interfaces/automotive/can/1.0/default/

regulations cover aspects such as guaranteeing that an over-the-air (OTA) update does not affect the safety certification of a vehicle. Furthermore, the regulations enforce the obligation for regular safety updates. While the standard does not mention fuzzing explicitly, it obligates the manufacturers to ensure software stability. Fuzzing is a good option for testing software stability.

In conclusion, automotive industry policymakers think cybersecurity is also crucial in the automotive sector. The regulations mentioned above make the manufacturers of automotive systems aware of cybersecurity risks in the field.

3 Related work

This section describes related work and resources on automotive security and novel fuzzers.

3.1 Security in automotive

Researchers have studied the security aspects of different protocols, technologies, and systems used in automotive. These studies have frequently revealed significant vehicle security issues, such as insecure protocols and privacy concerns. Our research focuses on Android Automotive and CAN interface, but we highlight why cybersecurity in the automotive sector is critical by illustrating various attacks and vulnerabilities.

In a study by Kyounggon Kim et al. [5], the authors have surveyed 151 papers related to automotive cybersecurity published from 2008 to 2019. The studies in their review show successful attacks against cars from Tesla, BMW, Ford, and Toyota. These papers describe the exploitation of different automotive systems, ranging from radio interference to zero-day vulnerabilities in WiFi and web browsers.

Some papers surveyed by Kyounggon Kim et al. and other studies [5, 49, 50] expose privacy and security concerns about the Tyre Pressure Monitor System (TPMS). Because the sensors and TPMS ECU in a car communicate without encryption, special equipment can easily read each sensor's unique ID, potentially allowing the tracking of vehicles based on these IDs. In these papers, authors often describe how TPMS ECU failures can occur due to man-in-the-middle (MiTM) and denial of service (DoS) attacks.

In addition, other papers surveyed by Kyounggon Kim et al. [5] indicate several security flows of the CAN protocol, making the protocol outdated and susceptible to attacks such as DoS, MiTM, and spoofing. Attacking the CAN protocol has dangerous consequences, given the role of the protocol in communicating with critical vehicle systems like brakes, airbags, and fuel injection systems. Attackers can execute these attacks remotely or locally by utilizing a malicious hardware device connected to the OBD2 port of a car.

The vulnerability of "keyless" systems to amplification and spoofing attacks is also a concern. Such attacks allow attackers without a physical key to unlock or start cars from a much greater distance than intended, provided the key is within range [5, 51].

Other researchers study the potential impact of specific Bluetooth stack vulnerabilities on vehicles equipped with Bluetooth-enabled infotainment systems. For example, Antonioli et al. [52] demonstrate that attackers can use BIAS (Bluetooth Impersonation AttackS) [53] and KNOB (Key Negotiation of Bluetooth) [54] attacks against in-vehicle Bluetooth devices of various famous carmakers such as KIA, Toyota, Suzuki, and Skoda.

Other authors' teams focus on the spoofing attack susceptibility of LIDAR-based ADAS sensors used in various vehicles currently. They present attacks that allow the creation of fake "cars" using specially crafted pixel points, triggering unnecessary sensor responses, such as emergency braking, despite no imminent danger [55].

As our research focuses on the Android Automotive OS, we need to discuss the security research already done about this OS. Researchers present their privacy concerns regarding inter-component communication (ICC) among apps in the ecosystem of Android Automotive. First, malicious apps may exploit the OS's API system to gather and leak information, such as vehicle identification details, location, and speed. Second, abusing the permissions of the apps can lead to damage to the vehicle components. Finally, an incorrect implementation by the IVI vendor may lead to arbitrary CAN injection [10, 15].

3.2 Commonly-used fuzzers

This subsection overviews the literature about common fuzzers, such as AFL and Angora. The fuzzers presented below are some of the most discussed, well-known, and used in the literature and production. In addition, researchers use most of them as a basis for further research in fuzzing technologies.

AFL(++) The American Fuzzing Loop (AFL) [20] is a well-known mutation-based grey box fuzzer, along with its more advanced iteration AFL++ [21], which is driven by the community.

AFL mutates test cases to discover new execution paths based on coverage-based feedback. Moreover, AFL applies optimizations such as minimizing test case size without impacting program behaviour. The two types of mutations used by AFL are deterministic and havoc. Deterministic mutations include single input changes like bit flipping, additions, and substitutions, while havoc mutations involve randomly adding, deleting, and altering inputs.

For further optimization, AFL uses a forkserver, which avoids the entire reinitialization of the program being fuzzed for faster execution of test cases. One more optimization used by AFL is Persistent Mode, where each iteration within

a loop uses one test case run at a time. Furthermore, AFL++ introduces other optimization approaches, such as prioritizing low-frequency paths to enhance coverage with its technical support, including LLVM, GCC, QEMU and Unicornafl, which are particularly useful for binary fuzzing.

Angora Angora [56] is a fuzzer increasing coverage without symbolic execution but still utilizing path constraints.

The main optimizations of Angora are as follows:

1. Context-sensitive branch coverage - increases coverage through executing path constraints in multiple contexts
2. Scalable byte-level taint tracking - reduces exploration space by tracking the input bytes affecting the path constraint and only mutating those that were modified
3. Search based on gradient descent - path constraint solver based on machine learning that increases the effectiveness of the fuzzer.
4. Type and shape inference - identifies input bytes used together, determines their data type, and enhances the mutation strategy of the fuzzer.
5. Input length exploration - increases input length if needed to trigger different paths, effectively increasing coverage.

Driller Driller [57] is a fuzzer that combines AFL fuzzer with angr, a symbolic execution engine. Driller introduces the concept of compartments, i.e., identifying and categorizing specific inputs since each may lead to a distinct execution flow. Driller employs concolic execution, i.e., running a program with symbolic execution techniques using concrete input data, which helps find as many execution paths as possible without affecting the fuzzing performance. The goal is to combine the strengths of fuzzing and symbolic execution.

The main Driller components are as follows:

1. Input test cases - no input test cases are required, but having test cases can speed up the initial fuzzing step by guiding the fuzzer toward specific compartments.
2. Fuzzing - execution of fuzzing engine when initially started, which explores the first compartment of the application until it reaches the first complex check on a specific input. Subsequently, the fuzzing engine stalls and cannot identify inputs to search for new paths in the program.
3. Concolic execution - activating its selective concolic execution component upon stalling the fuzzing engine. This component restricts user input to the unique inputs discovered in the preceding stage of fuzzing. The constraint-solving engine of the concolic execution component then pinpoints inputs that could force execution down unexplored paths.

4. Repeat - when the concolic execution component identifies additional inputs, the fuzzing component resumes mutation on these inputs to fuzz the newly identified compartments. Until discovering a test input that causes an application crash, Driller alternates between fuzzing and concolic execution.

VUzzer VUzzer [35] is an evolutionary smart fuzzer without using symbolic execution but using AFL-like principles. When mutating inputs, VUzzer prioritizes deep paths in the software and deprioritizes frequently traversed paths. Also, the fuzzer identifies where and how to alter the test inputs.

VUzzer contains components that work together to deliver an efficient fuzzing process that uncovers vulnerabilities through rigorous testing, as follows:

1. Dynamic Taint Analysis - the core component of the fuzzer tracks input flow in a program while identifying relevant memory locations and registers.
2. Magic-Byte detection - recognizes various input types (e.g., a valid JPG file) by using fixed byte values at specific offsets in the input, which helps generate valid input types for testing.
3. Basic Block Weight Calculation - the fuzzer rewards achieving difficult-to-reach code segments within nested control structures for optimal coverage.
4. Error-Handling Code Detection - assists in deprioritizing inputs leading the program towards an error code.
5. Fitness Calculation - similar to AFL, but the basic block weight determines if some paths are more attractive than others.
6. Input Generation - there are two types of input generation: crossover - which generates new child inputs from two-parent inputs, and mutation - which directly changes existing inputs.

QSym QSym [58] is a hybrid fuzzer that incorporates novel fuzzing techniques and symbolic execution. The fuzzer features a fast concolic execution engine with improved emulation performance, increased repetitive testing efficiency, concrete environment efficiency, and novel heuristics explicitly designed for hybrid fuzzers. Its authors claim that it surpassed Driller in performance during testing.

QSym has four major features:

1. Instruction-level execution - compared to other concolic execution engines, QSym only executes specific instructions at the instruction level when compared to other engines
2. Concrete environment modelling - uses concrete values to model external environments and avoids solving incompletely generated problems.
3. Optimistic Solving - attempts to solve only a part of a path constraint if it is incapable of entirely solving it.

4. Basic block pruning - eliminates redundant work by identifying similar basic blocks and utilizing only one block for generating path constraints.

libFuzzer LibFuzzer [22] is an evolutionary, in-process, coverage-guided fuzzer. The main features of libFuzzer include the following:

1. Coverage-guided fuzzing - compiler instrumentation that tracks each execution of code blocks with different inputs, monitoring the execution of different code sections for each input.
2. In-Process fuzzing - linking the target code to the fuzzer and executing both in the same process provides performance benefits due to decreased inter-process communication overhead.
3. Evolutionary input generation - libFuzzer uses a genetic algorithm to evolve the inputs of the target function.
4. Dictionary support - storage of relevant inputs in a dictionary for increasing the efficacy of fuzzing.
5. Value-Profile-Guided fuzzing - extends the coverage-guided fuzzing for measuring specific values variables can take, leading to more comprehensive target testing.

Comparison of the presented fuzzers By comparing the fuzzers presented in Section 3.2, we divide the fuzzers into two categories: fuzzers that use symbolic execution and fuzzers that do not use symbolic execution. Driller, VUzzer, and Qsym represent fuzzers that use symbolic execution, while AFL(++), libFuzzer and Angora do not use symbolic execution.

Even if we categorize these fuzzers, there are still differences among them. Driller uses fuzzing and symbolic execution to reach hard-to-reach code sections. VUzzer uses techniques of dynamic taint analysis, magic-byte detection, basic block weight calculation, error-handling code detection and fitness function inspired by AFL. QSym employs instruction-level concolic execution and concrete environment modelling. AFL uses coverage feedback and optimizations, such as minimizing test case sizes. Angora executes path constraints in different contexts and tracks input bytes that affect the path constraint. Finally, libFuzzer uses in-process fuzzing and evolutionary input generation.

3.3 Fuzzing technologies in automotive

Researchers have come up with solutions for fuzzing automotive systems in recent years. These solutions include fuzzing in a hardware-in-the-loop environment using identical test hardware components – one for fuzzing and one as a reference to contrast the behaviour of the fuzzed device. To be more detailed, the researchers have two identical vehicle dashboards: fuzzed and running normally. If the fuzzed dashboard displays malfunction lights, the researchers know that the

fuzzer has encountered an input that triggered a system crash. This experiment indicates errors like system crashes and losing control of systems [59].

In addition, Radu et al. [60] develop a grey-box approach for fuzzing ECUs using control-Flow Graph extraction from the firmware, which leads to some crashes in the units under test.

Furthermore, researchers fuzz essential protocols used in the automotive industry, such as CAN. By crafting malicious CAN packets, "Malfunction Indicator Lights (MIL) illumination, warning sounds and erratic gauge needles" occur in a test vehicle while fuzzing the interfaces [19, 61, 62]. Additionally, other research papers describe using fuzzers like beStorm [63], Defensics [64], CANoe/boolFuzz [65], and Peach [33] for research related to CAN protocol fuzzing.

Also, researchers adjust traditional fuzzing tools such as beStorm [63] to work with the improved version of CAN, CAN-FD, which enables higher throughput rates. However, Nishimura et al. [41] focus more on processing times than discovering new vulnerabilities or bugs using fuzzing.

Besides the academic approach, commercial tools for automotive security testing, including fuzzing, are available for automotive makers. Huracan by Riscure [66], an automotive security tool, enables developers to perform ECUs fuzzing. Moreover, the automotive industry can also use fuzzing-as-a-service options, such as Block Harbor [67].

4 Why the CAN interface?

In our research, we look at the CAN interface and explain why in this section. Android Automotive has many input interfaces of interest regarding fuzzing. However, we choose only one interface, the CAN interface, for our research because our time is limited. Therefore, we made a ranking for the most relevant interfaces in the automotive industry and Android Automotive environment in Table 1.

The importance of these interfaces mentioned in Table 1 is determined by their relevance for the fuzzing process. We have assessed their risk of security vulnerabilities and applicability to the automotive systems realm.

The top choice for our research is the Controller Area Network (CAN) interface, which plays a crucial role in the communication between Electronic Control Units (ECUs) in modern vehicles. Previous academic research has highlighted the importance of securing this interface, making it a good candidate for fuzzing. However, its age makes it both interesting and not interesting for new research because modern vehicles still use this protocol. In addition, the simulation of a CAN bus in a virtualized environment takes more effort because we simulate devices that communicate on the bus, thus making the fuzzing process more challenging.

Furthermore, other interfaces, such as Vehicle Manager, Remote Access, and External Vision System (EVS), have critical data management and control functions. Still, their lower vulnerability to input attacks makes them less interesting for fuzzing.

Interfaces like WiFi, Bluetooth and GPS are more vulnerable to internal and external cyber-attacks. Still, given that they are not specific to automotive systems, they have been rated lower in fuzzing priority.

Finally, we rank Media Oriented Systems Transport (MOST) and Local Interconnect Network (LIN) the last due to their lower risk of remote cyber-attacks or specific implementation limitations in the Android Open Source Project (AOSP).

Table 1: Input interfaces in Android Automotive

Ranking	Interface	Why to fuzz it?	Why not fuzz it?	Existent fuzzer in AOSP
1	CAN	Responsible for critical communication between ECUs, research on security aspects has been done, commonly used, protocol-specific fuzzing tools exist, vehicle-specific interface	Remote/outside attack less likely, older technology, probably harder to fuzz using virtual environment	Yes
2	Vehicle Manager	Potential memory leak found using existent libFuzzer harness, manages and controls data like vehicle speed, fuel level, engine temperature, and tire pressure, vehicle-specific interface	Not entirely an input interface	Yes
3	Remote Access	Control car over a network remotely, vehicle-specific interface	Not entirely an input interface	Yes
4	EVS	Malfunction can disturb drivers, critical in self-driving cars, vehicle-specific interface	Probably harder to fuzz (requires 2D/3D maybe)	Yes

5	SV - Sound and Volume	Malfunction can disturb drivers, vehicle-specific interface	Not necessarily an input interface	Yes
6	Audio	Malfunction can disturb drivers, vehicle-specific interface	Not critical component	Yes
7	Occupant awareness	Privacy issues in case of a security breach, vehicle-specific interface	Contains only AIDL definitions, not critical component	No
8	WiFi/4G/5G	Newer technology, susceptible to remote/outside attacks, researchers have researched the security aspects already, protocol-specific fuzzing tools exist	Part of base Android project (not Automotive specific)	No
9	Bluetooth	Newer technology, susceptible to remote/outside attacks, researchers have researched the security aspects already, commonly used, protocol-specific fuzzing tools exist	Part of base Android project (not Automotive specific), older technology	Yes
10	GPS	Susceptible to remote/outside attacks, protocol-specific fuzzing tools exist	Part of base Android project (not Automotive specific), older technology	No
11	USB	Commonly used, susceptible to remote/outside attacks, research on security aspects have been done	Older technology, part of base Android project (not Automotive specific)	No

12	Ethernet	Newer technology, susceptible to remote/outside attacks, protocol-specific fuzzing tools exist, researchers have researched the security aspects already	Probably harder to fuzz using a virtual environment, older technology	No
13	MOST	Commonly used, vehicle-specific interface	Mainly used for non-critical applications, remote outside/attack less likely, probably harder to fuzz using a virtual environment, implementation in the kernel but not automobile-specific	No
14	LIN	Responsible for critical communication between ECUs, newer technology, vehicle-specific interface	Less commonly used, remote outside/attack less likely, probably harder to fuzz using a virtual environment, no implementation in AOSP	No

5 Methodology

We aim to discover how we can use fuzzers with Android Automotive and how to use fuzzing to detect software vulnerabilities in hardware input interfaces. Further, we present our way of pursuing these goals.

In our research, we use white-box fuzzing because we have access to the source code of our targets. In addition, if we fuzz a library and not a standalone target, we link the library and fuzzer to the library to guide the fuzzer, which we define as a fuzzing harness. Our methodology consists of the following parts:

1. Compilation of the target's source code using code instrumentation
2. Preparing the initial input seeds
3. Fuzzing process
4. Analyzing and reporting the finding of the fuzzing process

The first step is compiling the source code of the target using code instrumentation. Code instrumentation is necessary for compiling the source code of the

target before starting the fuzzing process. Each fuzzing engine has its way of performing this instrumentation. We integrate instrumentation and building steps into the environment's building configuration, where the target's fuzzing process occurs. If the fuzzing process occurs in an emulated environment, cross-compilation is required.

The second step is preparing the initial input seeds. We use entirely random inputs and random inputs based on the standard inputs the target usually uses in normal operations. The fuzzer needs these initial input seeds as a starting point for the fuzzing process.

The third step is the fuzzing process. While the fuzzing process is running, the fuzzer loops through the input seeds and constantly observes the behaviour of the target using these seeds. These observations mainly comprise checking the code coverage using each input and system abnormalities, such as crashes, timeouts and memory leaks.

Based on these observations, the fuzzer creates new input seeds using mutation, generation and evolutionary algorithms to increase further the code coverage and the chances of discovering new system abnormalities. If the newly-discovered input seed does not lead to a new path in the target or does not create instability in the system, the fuzzer discards the input seed. When the fuzzer finds an input seed that increases the coverage or crashes the target, the fuzzer uses this input as a base for further input generation. We design a fuzzing harness based on the original harness with several tweaks to improve the fuzzing performance. The fuzzer records its performance and status in log files and the inputs that make the target crash.

The fourth and last step is analyzing and reporting the finding of the fuzzing process. When the fuzzer completes the fuzzing process, we check the input seeds that the fuzzer considered that made the target unstable, if there are any. We run the target using these inputs to confirm whether these inputs make the target unstable.

6 Implementation

This section discusses how we implement fuzzing in the AOSP environment and prepare the fuzzing experiments.

6.1 Android Open Source Project (AOSP) setup

Before we perform any fuzzing experiments, we need to download and build the AOSP source code. After downloading the source, we compiled the code source using a specific configuration for Android Automotive present in AOSP. We use the AOSP product configuration `sdk_car_portrait_x86_64-userdebug`. We choose a configuration that allows us to emulate the x86_64 architecture and control entirely the emulator system, e.g. root access and full R/W rights on the system partitions.

6.2 Running and using AOSP emulator

After successfully compiling AOSP from sources, we run the Android Emulator using the following options: enable writable `/system` partition and disable showing the emulator window. We want to have complete R/W control over the emulator. In addition, we do not use the emulator's window because we run our commands using the terminal, and our server's resources are limited.

We provide `root` privileges to the emulator for full access to the system files. In addition, we need to remount the system partitions on the emulator and sync the files on the emulator.

The next step we take is running multiple emulators in parallel to make our research more efficient. We use the `avdmanager` and `sdkmanager` tools to manage this. Firstly, we download a system image with similar specifications (UpSideDownCake version, x86-64) using `sdkmanager` and then copy the `.img` files from our compiled AOSP emulator images. Then, we create the necessary emulators using `avdmanager create avd` command. Finally, as stated above, we use the `emulator` command, but we also specify what AVD we use by adding the parameter `@avd_name`.

6.3 Choosing the fuzzing frameworks used in the research

We focus on the following fuzzing frameworks in this research: libFuzzer and AFL++ (referred further simply as "AFL"). We present the technical and scientific considerations that guided us in our choice.

We do not use fuzzers that use symbolic execution due to the high resources necessary for running them. Android Automotive is usually running on limited hardware, and also, due to limited virtualization capabilities, we are concerned that our setup can not handle these fuzzing engines.

On a high level, these two fuzzers have different ways of functioning. AFL is appealing because it works with an instrumented binary of the target and typically requires manually writing a harness in the form of writing main alternative functions. However, we can run AFL without performing source code modifications. Having the source allows us to understand the code better and tweak the fuzzing process. LibFuzzer requires a harness which calls the specific functions linked to the original code using the compiler's linker.

Furthermore, initial input data requirements differ for each fuzzer: AFL requires pre-existing data, but libFuzzer can start without it. Therefore, we generate initial input data for a meaningful comparison between experiments, regardless of libFuzzer's capability to run without initial data.

In our experiments, we use white-box fuzzers. Thus, LibFuzzer and AFL can both deal with white-box systems. Furthermore, the fuzzers use mutation, genetic and evolutionary techniques to create input seeds which fit our requirements.

Technically, AOSP developers have integrated libFuzzer and AFL into the AOSP build system, which helps set up our research environment. However, we have encountered issues while setting up AFL for our experiments. Also, both fuzzers can fuzz the C/C++ code in our harnesses.

Finally, we only choose these two fuzzing frameworks because choosing more increases the research complexity and our time is limited for this research. Adding other fuzzers needs complex changes to the Android build system. The fuzzers we select are already in the AOSP ecosystem. Even though AOSP integrated AFL, we still have to change the build files to compile the harnesses because of compilation errors caused by duplicate symbols during the linkage part of the compilation. We have spent considerable time developing a fix for this issue, described in Section 6.4. Thus, integrating a fuzzer that has never been integrated into the AOSP ecosystem requires even more time and effort.

6.4 Fuzzing harness compilation and running

The fuzzing library harnesses, part of the AOSP source code, are designed explicitly for libFuzzer, and we notice developers have written the hardware interfaces implementations in C/C++.

In the latest versions of the AOSP, including the one we use, we can configure the fuzzing framework by just setting the `FUZZ_FRAMEWORK` environmental variable to `libfuzzer` or `afl`. If we do not set the variable explicitly, it defaults to using libFuzzer. Additionally, AFL provides a driver that instruments existing libFuzzer harness targets for fuzzing using AFL. However, the AFL integration is incomplete, and we have to modify the build files, as shown in Appendix A.1.

Before starting the fuzzing process, we manually populate the "inputs" folder with initial corpora. After completing this step, we start the fuzzing process.

When we use libFuzzer, we use fork mode while running libFuzzer because we do not want the fuzzer to stop when a crash is detected. Furthermore, we disable the memory leak detection in libFuzzer because it interferes with the fuzzing process. In some of our cases, a memory leak causes the fuzzer to detect a crash right at the start, preventing any further fuzzing from being conducted. Furthermore, memory leaks are not generally a severe security vulnerability.

When using AFL, we use the default settings, except the timeout set to 5000ms and enable the deterministic mode for fuzzing for increased fuzzing efficiency. The deterministic mode is more efficient because it does not repeat test cases, has a systematic approach to increasing code coverage, and the results are easier to reproduce.

6.5 Modified CAN interface harness

Our experiments use a modified version of the original CAN interface fuzzing harness shipped with the AOSP source code. In Table 2, we present for which

libraries we introduced direct calls in the modified version and the reasons for these choices. In this modified version, we have introduced direct calls to the following libraries: `CanController`, `CanBusNative`, `CanBusSlScan`, `CanBusVirtual`, and `CanSocket`. We omit the `CanBus` library, as the other libraries have already called all its relevant functions. Based on the call graph shown in Figure 2 and manual source code inspection, we have decided on what direct calls we add in the modified harness. We present the call graph of the modified (improved) harness in Figure 3.

Table 2: List of added direct calls in modified harness

Library name	**Function**	**Reason to include direct calls in the improved harness**
CanController	getIfaceName	Not directly fuzzed, but it might get fuzzed data. The function takes the path to a serial interface in `/sys` [68] as a parameter.
	readSerialNo	Not directly called and fuzzed in the original harness. Takes the path to serial interface from `/sys/devices/` as a parameter [68].
	findUsbDevice	Called but not fuzzed. The parameter of the function represents "a list of serial number (suffixes) from the HAL config" [68].
	getSupportedInterfaceTypes	Not directly called and fuzzed in the original harness. The function takes as a parameter a return callback [68].
	isValidName	Not directly called and fuzzed in the original harness. The parameter is a simple string [68].
CanBusSlcan	All	Not called directly. Most functions take the following parameters interface's name on the system as a string and the bitrate as a 32-bit unsigned integer [68].
CanBusVirtual		
CanBusNative		
CanSocket		Not directly called and fuzzed in the original harness. The functions here usually take parameters such as callbacks, interface names, and CAN frames [68].

AOSP developers have already included the `CanController` library in the harness. Still, we include direct calls to the following helper functions: `findUsbDe-`

`vice`, `readSerialNo`, `getIfaceName`, and `isValidName`. We choose these functions because they are not directly fuzzed in the original harness but might get called using fuzzed data. To make these functions accessible to our harness, we modify the source code of `CanController.cpp`: we remove the `static` keyword. We add the definitions for these functions in the header file `CanController.h`.

The following library brought into discussion is `CanBusNative`. This library is not originally directly included in the harness, but we introduce a direct call to this library to its `preUp` function. Similarly, we proceed with the libraries `CanBusVirtual` and `CanBusSlcan`.

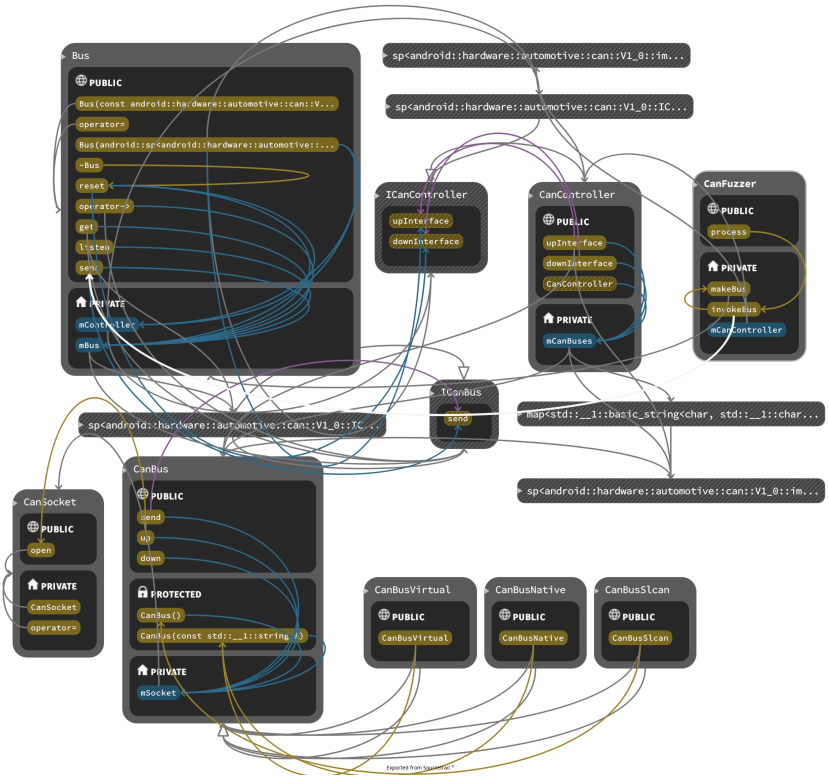

Fig. 2. Call graph of the original harness

Lastly, we introduce more direct calls for the `CanBusSocket` library because the harness does not directly include it. The functions included in this library might benefit from direct calls with fuzzed data, so we potentially increase the coverage. We modify the header file to have more direct access and remove the private keyword for the functions defined. Firstly, we define two mock callbacks

in the header file: `ReadCallback` and `ErrorCallback`. Then, we open a CAN socket. If we successfully open the CAN socket and there is still available fuzzed data, an object with fuzzed data is sent to the opened CAN socket.

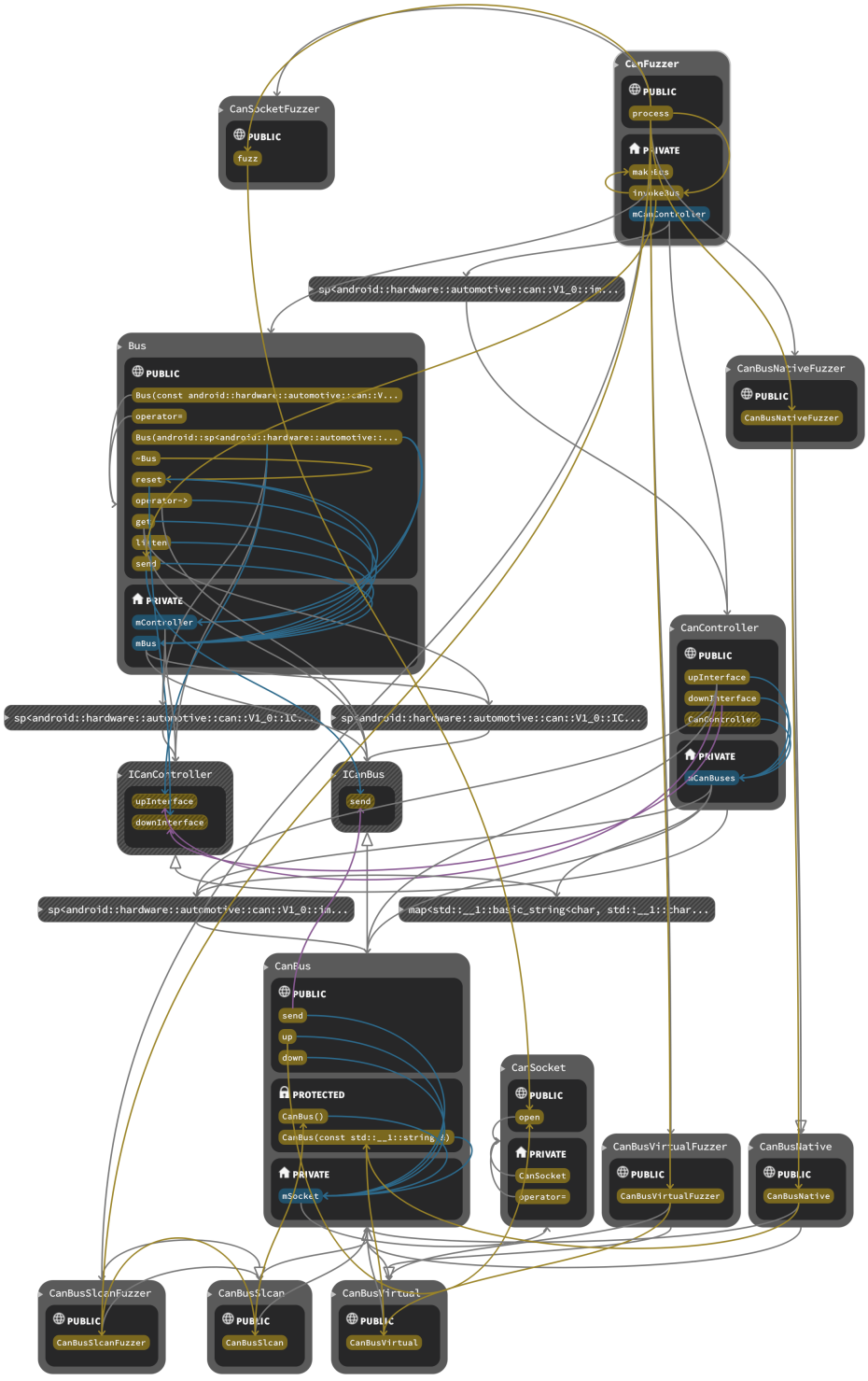

Fig. 3. Call graph of the improved harness

6.6 Initial corpora data

We need initial corpora data, also known as "seed inputs", to run the fuzzing experiments. This corpora data is essential because they provide a starting point, especially for AFL. In our experiments, we define two types of corpora data: simple and complex.

The simple initial corpora data consists of the string `A`. We choose this string because it is simple enough to start the fuzzing process with libFuzzer and AFL.

We also define complex initial corpora data that consists of the three files representing different CAN message types: standard CAN message, CAN FD message, and a message that only represents a Remote Transmission Request (defined in the source code). We create these inputs based on the variables and structures of the original harness. Each file contains the following messages:

- `id` - CAN message that is a standard ID (11 bits) or an extended ID (29 bits), represented in HEX
- `payload` - actual data carried by a CAN message, which has a length between 0-8 bytes for standard CAN, and up to 64 bytes for CAN FD, represented in HEX as well
- `timestamp` - time since boot measured in nanoseconds
- `remoteTransmissionRequest` - a boolean variable that determines data retrieval request from another CAN network ECU, which, when it is set to 0, there is no payload set
- `isExtendedId` - sets either the message using standard ID (value is 0) or extended ID (value is 1)

Below, we present the actual content of these three files:

- Standard CAN message, 11-bit ID, 8-byte payload:

```
id: 0x012F
payload: 01 02 03 04 05 06 07 08
timestamp: 0
remoteTransmissionRequest: 0
isExtendedId: 0
```

- CAN FD message, 11-bit ID, 15-byte payload:

```
id: 0x012F
payload: 01 02 03 04 05 06 07 08 09 0A 0B 0
    C 0D 0E 0F
timestamp: 0
remoteTransmissionRequest: 0
isExtendedId: 0
```

- Remote Transmission Request, 11-bit ID, no payload:

```
id: 0x012F
payload:
timestamp: 0
remoteTransmissionRequest: 1
isExtendedId: 0
```

6.7 Fuzzing metrics measurement

We re-run the corpora generated by the experiments using the libFuzzer through the `afl-showmap` tool to measure the coverage using the AFL-instrumented binary. In addition, we do the same with AFL-generated corpora. We perform this operation because libFuzzer does not provide comparable coverage data with AFL.

The upside of using `afl-showmap` for AFL and libFuzzer results is that the coverage results are comparable across fuzzers. The downside of this uniformization of results is that libfuzzer may do great at optimizing its coverage according to libfuzzer, and it looks bad when using `afl-showmap` because AFL uses a different way to measure coverage.

6.8 Emulators provided by the car manufacturers

Besides the emulator we compiled from the AOSP source code, we run experiments on emulators offered by car manufacturers. We have discovered that General Motors (GM), Volvo, Polestar, and Honda offer these for developers. In our research, we use the ones using Android 11 from GM and Honda. We only use Android 11 or higher emulators because, upon source code inspection, the others do not come with a CAN hardware interface source code.

We present the car emulators used in our research below:

- GM[2]
  - MY24 CADILLAC Lyriq Freeform SUV[3]
  - MY24 GM SUV[4]
- Honda[5]

We also need root access to have complete control over the emulators provided by car manufacturers. We can easily do that using ADB commands in the standard Android Automotive emulator we compile directly from the source code. However, the car manufacturers build the provided emulators in production mode, so the `adb root` command is blocked.

[2] https://developer.gm.com/docs/gm-emu-downloads (requires account creation)
[3] https://developer.gm.com/downloads/final_ff_05302022_emulator.xml
[4] https://developer.gm.com/downloads/final_31XX_06272022_emulator.xml
[5] https://global.honda/cars-apps/index.html

We root the virtual images using the rootAVD[6] tool. The instructions on installing and using the tool are present there, including some special notes for Android Automotive images and devices with QEMU images. In addition, we configure Magisk, so we always grant root access without showing the SuperSu prompt.

Compared to the original AOSP emulator image, we enable root access by using the `su` command in the terminal shell of the emulator. In addition, we cannot use the commands `adb remount` and `adb sync` with the rooted emulators: we first need to copy the files to a location on the emulator freely accessible, and then use the shell with root rights, we copy them to our location of choice.

Furthermore, the CAN interface implementation in Android 11 does not have a fuzzing harness. Thus, we use the original harness used in the experiments with the AOSP emulator, and we adapt the source code and build files of the harnesses to match the implementation of the CAN interface in Android 11. We have replaced the calls to features not present in the Android 11 Native Development Kit (NDK). The source code is available in the Appendix A.

7 Experiments

This section covers the experimental setup, experiments, and their results.

For our first experiment, we evaluated libFuzzer and AFL performance on the default and improved harness using the simple input corpus. We used libFuzzer and AFL to test both the default and upgraded CAN interface harnesses for 24 hours, five times, on the original AOSP emulator. Our experiment test Hypothesis 1, which states that harness complexity correlates with code coverage and the number of crashes detected in the source code. We expect AFL to outperform libFuzzer in discovering crashes.

The second experiment repeats the previous one, wherein the default and improved CAN interface harnesses are run five times for 24 hours on the original emulator built from AOSP sources, using libFuzzer and AFL. The distinction in this experiment is that we use the complex corpus presented in Section 6.6. Using this experiment, we test Hypothesis 1 and Hypothesis 2. With these hypotheses, we aim to show that increasing the harness's complexity also increases the source code's code coverage or the number of crashes found and that libFuzzer detects fewer crashes than AFL.

The final experiment comprises running the first and second experiments on GM and Honda's emulators. These emulators use different Android versions compared to the original AOSP emulator. During these experiments, we test Hypothesis 1 and Hypothesis 3. With these hypotheses, we aim to prove that the number or type of crashes is influenced by the Android version, besides the experimental goals of the first and second experiments. However, due to the lack

[6] https://github.com/newbit1/rootAVD

of AFL integration in the Android 11 build system, we do not test Hypothesis 2.

7.1 Experiment set-up

We perform the experiments on a Ubuntu 22.04.2 LTS server with the following specifications:

- CPU: 2 x Intel Xeon CPU E5-2620 v3 @ 2.40GHz, with 12 threads on each core
- RAM: 64 GB DDR4
- Storage units: 1 x Samsung SSD 850 and 2 x WDC WD4001FFSX-6

We perform the fuzzing processes on Android Automotive Emulators running Android UpSideDownCake codename with an x86-64 architecture, one CPU core and 2GB RAM. We chose x86-64 as the processor architecture because we have noticed that all the emulators published by the car manufacturers use either x86-64 or x86. In addition, we can only have emulators with a single virtual core because of the limited CPU virtualization features of our server and Android emulation ecosystem.

7.2 Running both default and improved CAN interface harnesses five times for 24h with both libFuzzer and AFL

Our first experiment consists of fuzzing the following harnesses: the original AOSP one and the modified harness. We use libFuzzer and AFL on the official AOSP emulator for this experiment. We run the compiled targets for libFuzzer and AFL for 24 hours, five times each, using the simple initial corpora data defined in Section 6.6, and we run the experiments on the server in 5 different instances of the original AOSP Android Automotive x86-64 emulators.

In Figure 4, we present our coverage measurements from fuzzing the default CAN interface harness and the modified ("improved") one with libFuzzer. Looking at the default harness coverage, we notice that libFuzzer achieved higher coverage than AFL. However, the situation is different when looking at the improved harness: AFL has a significantly higher coverage rate than libFuzzer. If we compare the results overall, we notice that the enhanced harness performs similarly or slightly worse than the default harness when we use libFuzzer as a fuzzing framework but has an outstanding higher coverage performance than the default harness when we use AFL. We think this happens due to the non-determinism of libFuzzer, which brings unpredictability and non-reproducibility in results.

The next metric we used in comparing the harnesses and the fuzzing framework is the number of detected unique crashes (according to AFL) for each harness and each framework, presented in Figure 5. In both the default and improved harness, libFuzzer fails to detect crashes. On the other hand, AFL manages to

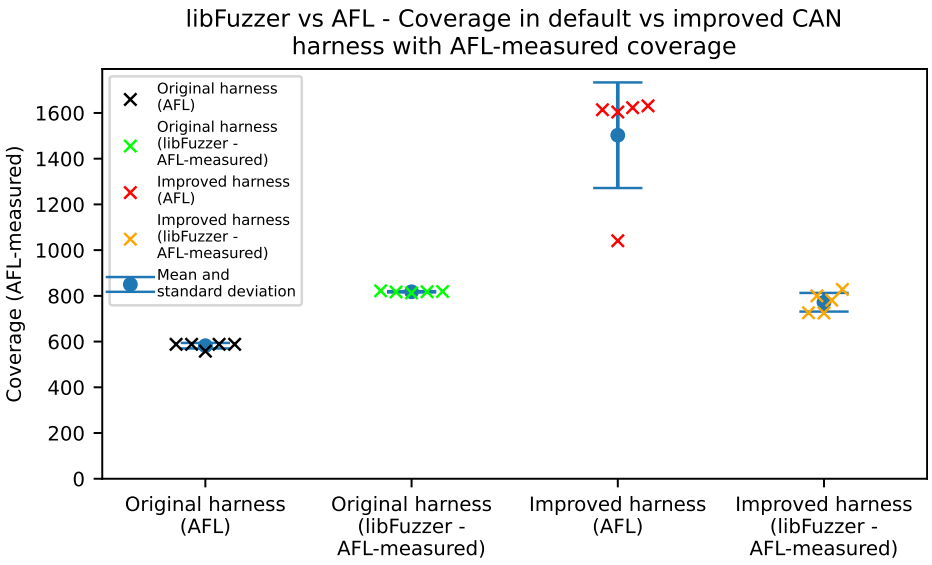

Fig. 4. libFuzzer vs AFL - Coverage in default vs improved CAN harness with AFL-measured coverage

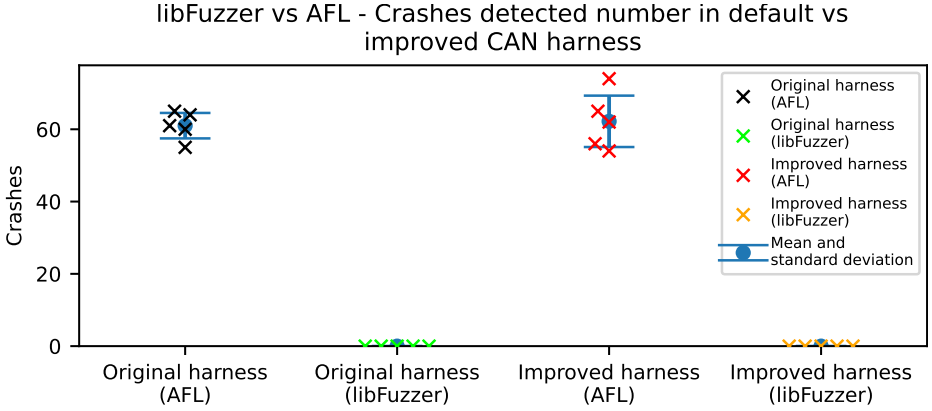

Fig. 5. libFuzzer vs AFL - Crashes detected number in default vs improved CAN harness

discover crashes using all of the harnesses. Furthermore, in 2 out of 5 runs, the improved harness has better results than the original one. By modifying the harness, we think AFL had to perform more operations within 24h than with the default harness. Furthermore, we have replicated some of the crashes found by AFL and all of them have a similar pattern:

```
external/libcxx/include/sstream:562:28: runtime error
    : implicit conversion from type 'int_type' (aka '
```

```
int') of value 255 (32-bit, signed) to type '
char_type' (aka 'char') changed the value to -1
(8-bit, signed)
```

Thus, they occur in the same place and maybe false-positive crashes. AFL considers these crashes unique because the tested and resulted values differ.

In addition, verbose logs of `AddressSanitizer` show memory leaks at the end of each fuzzing run. However, the fuzzer does not generate specific crash files to replicate these leaks. Thus, we think that these leaks are bugs of the `AddressSanitizer` running on the emulator.

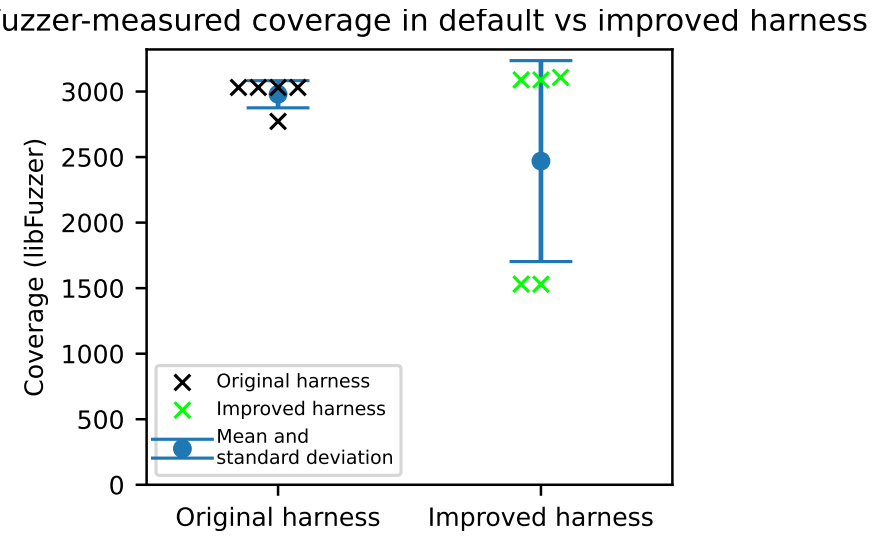

Fig. 6. libFuzzer-measured coverage in default vs improved harness

Also, we present a comparison of measured coverage using libFuzzer between the default harness and modified harness in Figure 6. In the case of libFuzzer, the coverage represents the number of code blocks or edges found. It is noticeable that in 3 runs out of 5, the improved harness has slightly better coverage than the default one. This overall performance increases due to the added calls in the improved harness. However, the lower performance in some of the runs is due to the non-deterministic behaviours of the fuzzer.

Figure 7 and Figure 8 shows the coverage evolution versus time. The data presented in 7 correlates with data shown in Figure 6 and confirms the abovementioned observations. In addition, when testing the default harness, one of the runs features a sudden increase in coverage around 20000 seconds timestamp, which occurs due to fuzzing non-deterministic behaviour. We also observe that

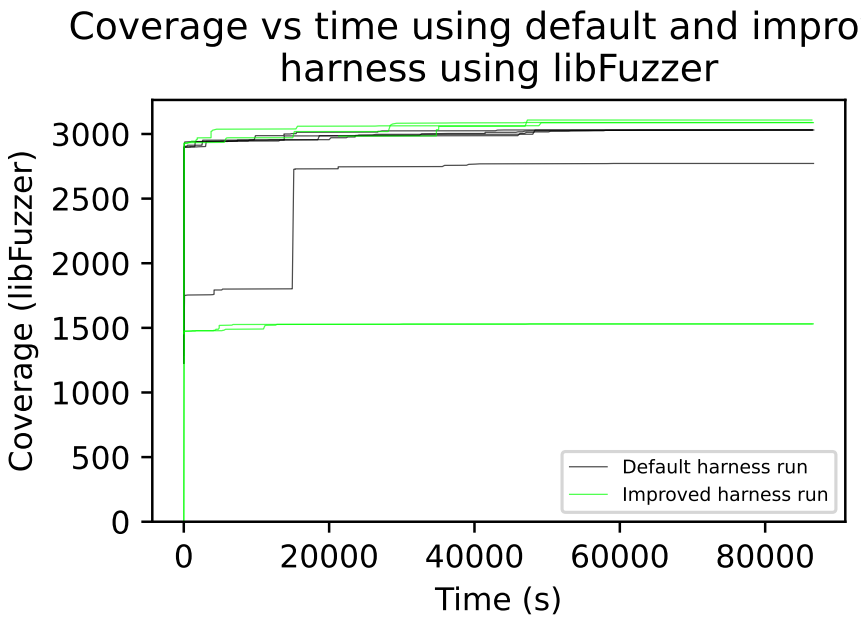

Fig. 7. Coverage vs time using default and improved harness using libFuzzer

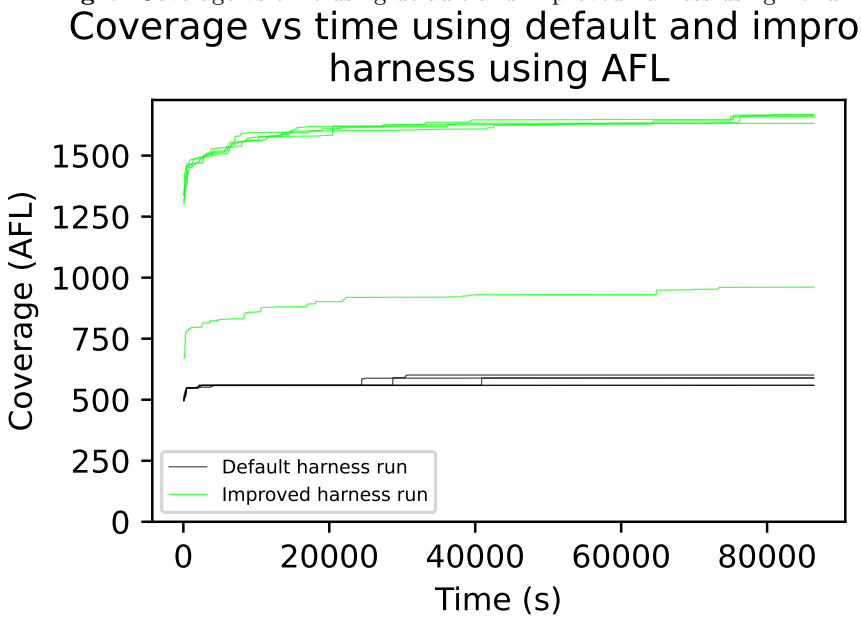

Fig. 8. Coverage vs time using default and improved harness using AFL

the libFuzzer achieves most of the coverage initially. Over time, most of the coverage curves saturate around 50000 seconds. We think that libFuzzer achieves such performances due to its coverage-focused algorithms. Thus, fuzzing for 24 hours may be too much in this case.

We also analyze the coverage evolution versus when using AFL in Figure 8. Again, the data presented in 4 correlates with data shown in Figure 4 and confirms the abovementioned observations. We observe that the coverage curve when using AFL is slow. AFL achieves these coverage curves with a more deterministic approach and improved fuzzing algorithms. In addition, the improved harness can benefit from running the fuzzer longer than 24 hours.

7.3 Use complex initial corpus and rerun experiment #1

The second experiment is similar to the previous one, but we attempt to use the complex initial corpus files defined in Section 6.6. Besides the different initial corpus, we run the experiments the same way as in the previous experiment in Section 7.2.

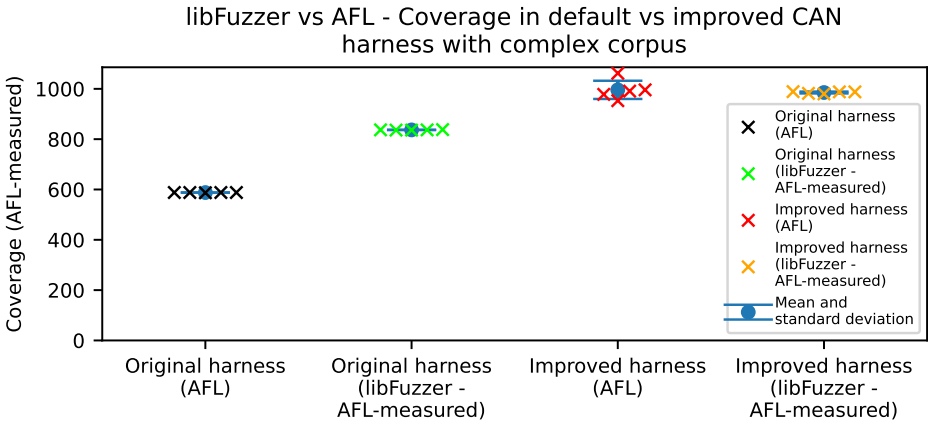

Fig. 9. libFuzzer vs AFL - Coverage in default vs improved CAN harness with complex corpus

Further, we compare libFuzzer and AFL using the default and improved harness in terms of coverage in Figure 9. The improved harness has better coverage than the default one for libFuzzer and AFL in all five runs.

When we look at libFuzzer-measured coverage in Figure 10, the improved harness outperforms the original harness in all of the runs. Furthermore, the standard deviation among the runs in the improved harness is higher than in the original. We think this happens because the higher complexity of the improved harness also makes the non-deterministic behaviour of the fuzzer more prominent.

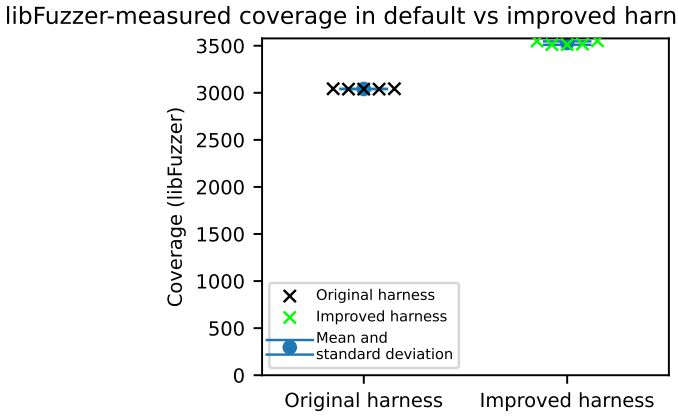

Fig. 10. libFuzzer-measured coverage in default vs improved harness - complex corpus

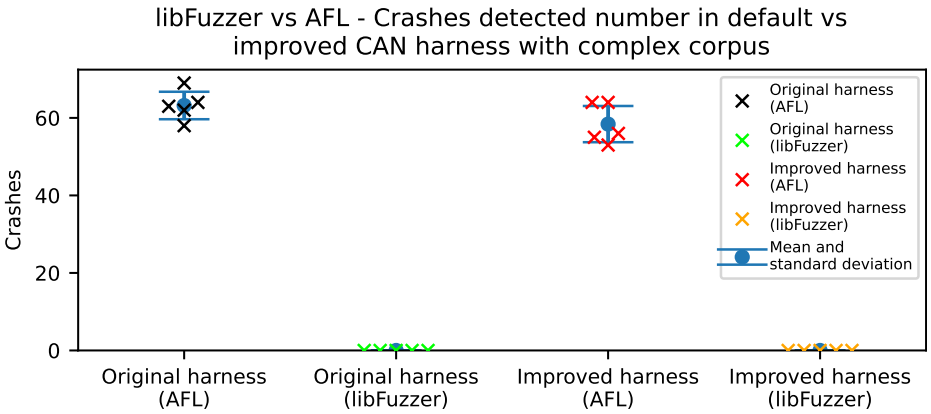

Fig. 11. libFuzzer vs AFL - Crashes detected number in default vs improved CAN harness with complex corpus

In Figure 11, we compare libFuzzer and AFL regarding the crashes detected number in default and improved CAN harness while we use the complex corpus. Again, libFuzzer fails to find any crashes. Another interesting observation is that AFL detects fewer crashes using the improved harness than the default one in 3 out of 5 runs.

Furthermore, we present coverage evolution versus time using the complex corpus in Figure 12 and Figure 13. The data presented in 12 correlates with data shown in Figure 10 and confirms the abovementioned observations. Furthermore, libFuzzer achieves most of the coverage initially. Over time, most of the coverage curves saturate around the end of the fuzzing seconds. We think that libFuzzer achieves such performances due to its coverage-focused algorithms. Finally, the

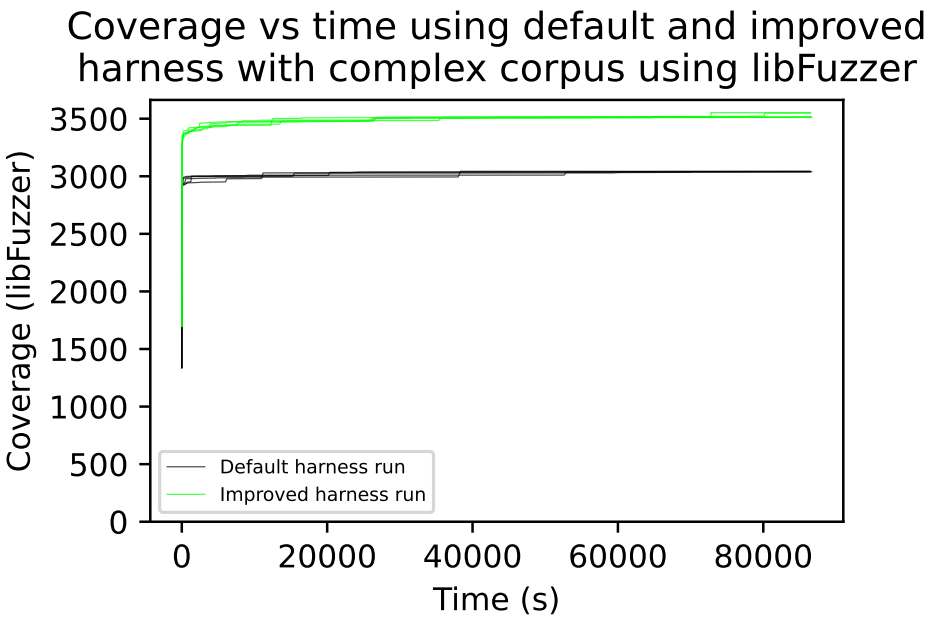

Fig. 12. Coverage vs time using default and improved harness with complex corpus using libFuzzer

improved harness can benefit from running the fuzzing process longer than 24 hours, but not much.

Using AFL in Figure 13, we also analyze the coverage evolution versus time. Results from Figure 9 correlate with results presented in Figure 9 and confirms our previous observations. We observe that the coverage curve when using AFL is slow. Especially when fuzzing the improved harness, we see that the curve still has ascending trend at the end of the fuzzing process. Thus, the improved harness can benefit from running the fuzzer longer than 24 hours. AFL achieves these coverage curves due to a more deterministic fuzzing approach.

7.4 Running fuzzers on different emulators from car manufacturers

The third experiment represents running Experiments 1 and 2 on emulator images provided by car manufacturers for developers, provided by GM and Honda. These emulators use Android 11. In addition, the build system does not include AFL in this version, so our experiments only include fuzzing with libFuzzer.

Regarding results, libFuzzer fails to detect crashes regardless of the Android version and emulator used. This failure might happen for potential reasons: libFuzzer's fuzzing mechanism, harness not reaching one or more vulnerable areas, improper corpus, or even faulty libFuzzer implementation in the Android ecosystem.

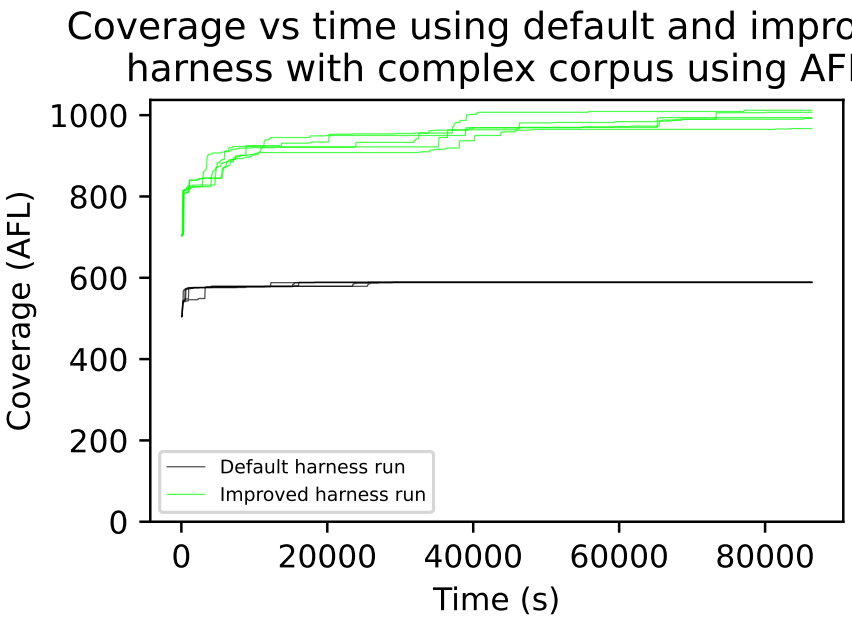

Fig. 13. Coverage vs time using default and improved harness with complex corpus using AFL

Honda We present the coverage measurements using libFuzzer in default vs improved CAN harness using the Honda emulator in Figure 14. The improved harness shows improved coverage over the original harness. However, when using the simple corpus, the standard deviation shows much difference between runs in the improved harness. The fuzzer's non-deterministic behaviour and using a less precise initial corpus can explain this difference. In addition, the standard deviation of the runs of the original harness shows that all the runs have achieved the same coverage. We think this happens because the fuzzer achieves the maximum achievable coverage of the harness or because the corpora generated do not lead to new paths in the harness.

In Figure 15, we present results of coverage vs time using the simple corpus on the Honda emulator. We observe that the improved harness coverage curves stabilize around 60000s. In comparison, the default harness curve stabilizes at around 20000 seconds. The improved harness takes longer to achieve curve saturation due to the increased complexity of the harness itself but also due to the more complex initial input. As a result, the improved harness can benefit from fuzzing for longer than 24 hours.

We inspect the coverage over time using the complex corpus on the Honda emulator shown in Figure 16. We observe that the improved harness coverage curves continue to increase towards the end of the fuzzing process. In comparison, the default harness curve stabilizes around 50000 seconds. Again, the improved

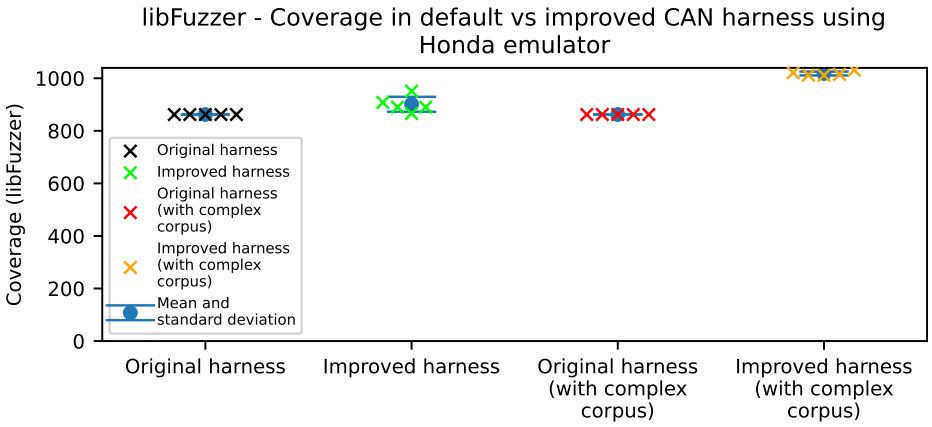

Fig. 14. libFuzzer - Coverage in default vs improved CAN harness using Honda emulator

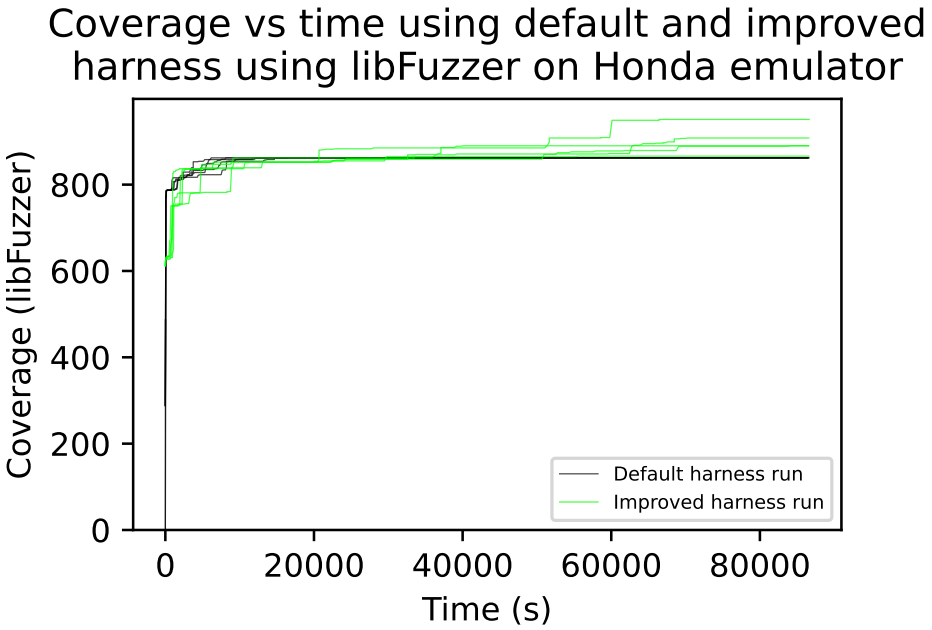

Fig. 15. Coverage vs time using default and improved harness using libFuzzer on Honda emulator

harness takes longer to achieve curve saturation due to the increased complexity of the harness itself but also due to the more complex initial input. As a result, the improved harness can benefit from fuzzing for longer than 24 hours.

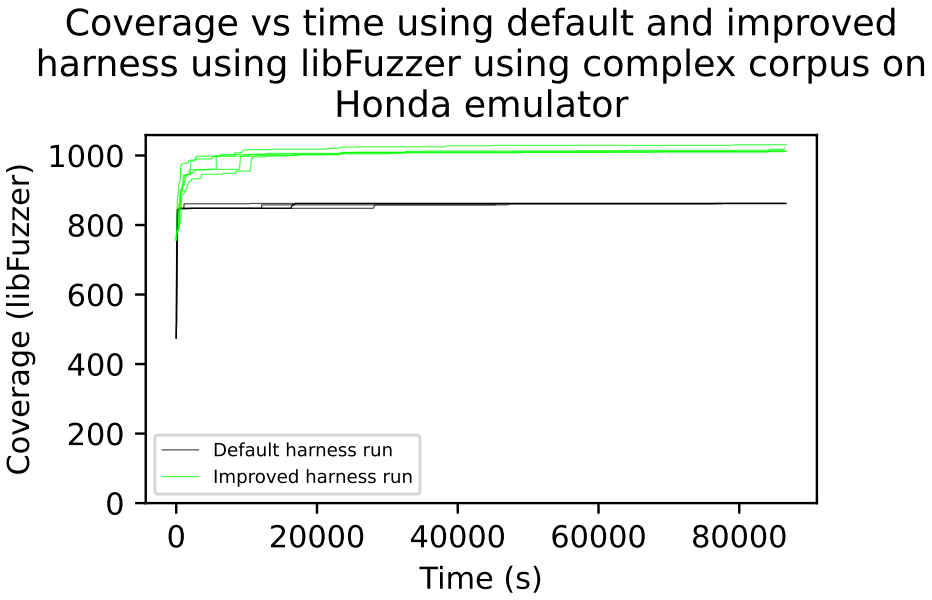

Fig. 16. Coverage vs time using default and improved harness using libFuzzer using complex corpus on Honda emulator

GM Lyriq When we observe the libFuzzer's results using the GM MY24 CADILLAC Lyriq Freeform SUV (abbreviated as GM Lyriq) emulator using simple and complex corpus, depicted in Figure 17, our comments are similar with the ones from the similar experiment performed using the Honda emulator in Figure 14.

We show the coverage over time using the simple corpus on the GM Lyriq emulator in Figure 18. We must mention that we have lost some data on three improved harness runs due to technical issues. The improved harness coverage curves saturate in the 50000 and 60000 seconds, while the default harness curves reach most of their limit around 20000 seconds. Again, due to increased complexity, the improved harness takes longer to achieve curve saturation and may benefit from fuzzing for longer than 24 hours.

We show the coverage over time when using the complex corpus on the GM Lyriq emulator in Figure 19. We must mention that we have lost some data on one of the improved harness runs due to technical issues. We observe that most curves saturate between 20000 seconds and 40000 seconds. The harness's

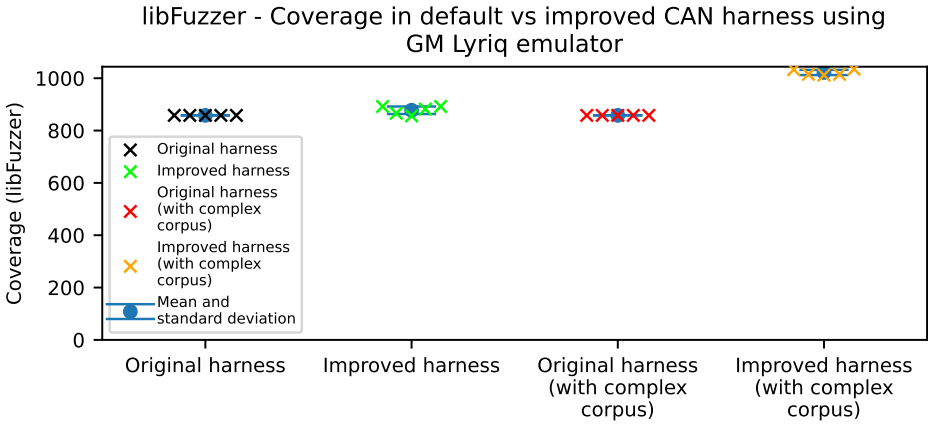

Fig. 17. libFuzzer - Coverage in default vs improved CAN harness using GM Lyriq emulator

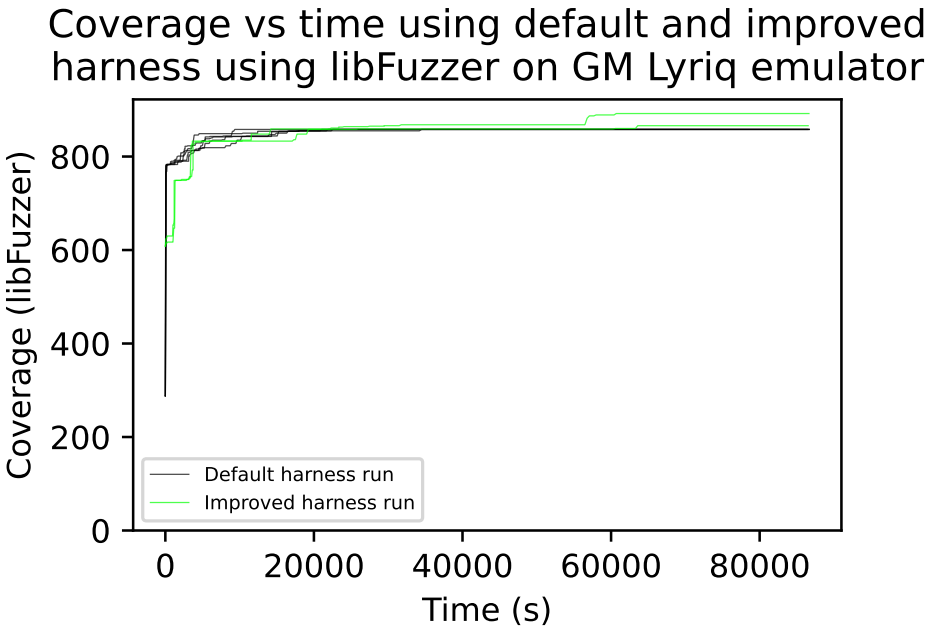

Fig. 18. Coverage vs time using default and improved harness using libFuzzer on GM Lyriq emulator

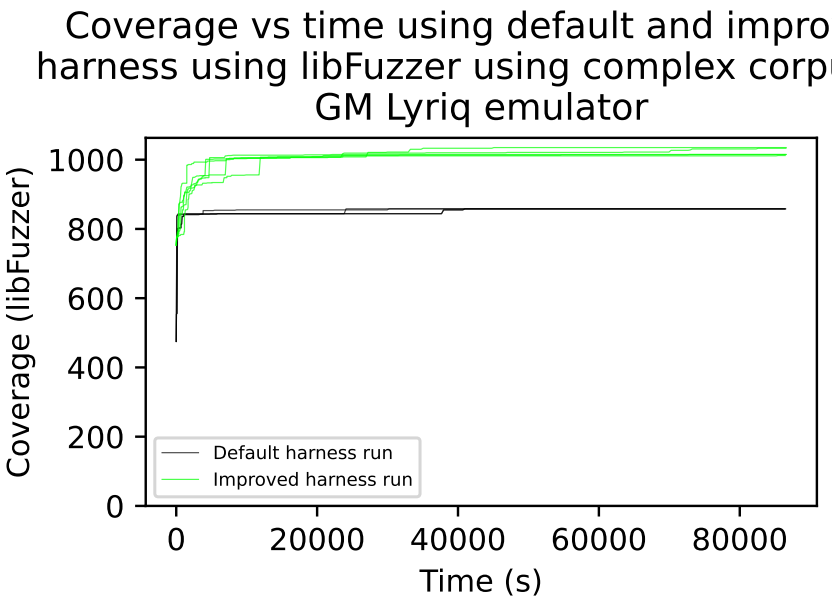

Fig. 19. Coverage vs time using default and improved harness using libFuzzer using complex corpus on GM Lyriq emulator

increased complexity and the more complex initial corpora make it harder for the fuzzer to reach maximum coverage. Again, the improved harness can benefit from fuzzing for longer than 24 hours.

GM SUV In Figure 20, we present the coverage results using the GM SUV emulator. We observe that the results of the original harness are very similar when using the both simple and complex initial corpus. Again, the improved harness results better when the complex corpus is used. Furthermore, the standard deviation is minimal in all the results presented in the figure.

Further, we look again at the coverage vs time when using the simple corpus on the GM SUV emulator in Figure 18. The improved harness coverage curves saturate close to 80000 seconds, while the last default harness curve reaches saturation at around 70000 seconds. Due to increased complexity, the improved harness takes longer to achieve coverage saturation and may benefit from fuzzing for longer than 24 hours.

Lastly, we present our coverage vs time results using the complex corpus on the GM SUV emulator in Figure 19. The default harness curves saturate at most around 65000 seconds into the process. Again, the last improved harness coverage curves saturate close to 80000 seconds. Again the improved harness can benefit from fuzzing for longer than 24 hours.

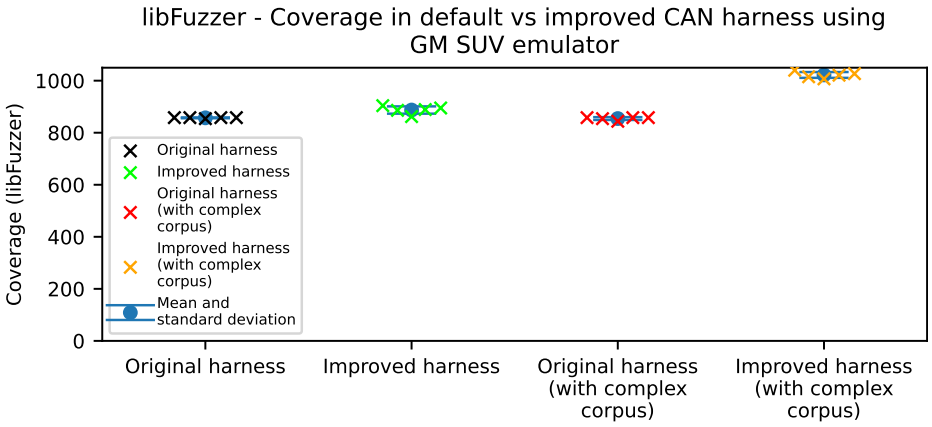

Fig. 20. libFuzzer - Coverage in default vs improved CAN harness using GM SUV emulator

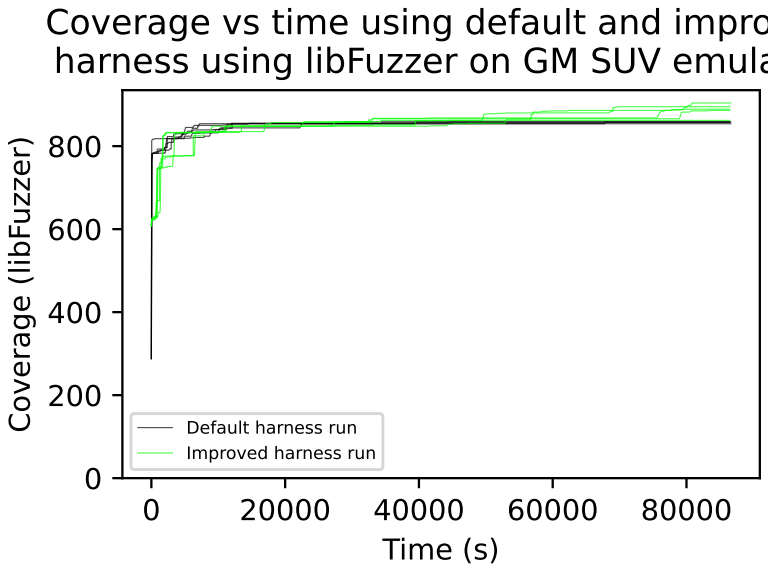

Fig. 21. Coverage vs time using default and improved harness using libFuzzer on GM SUV emulator

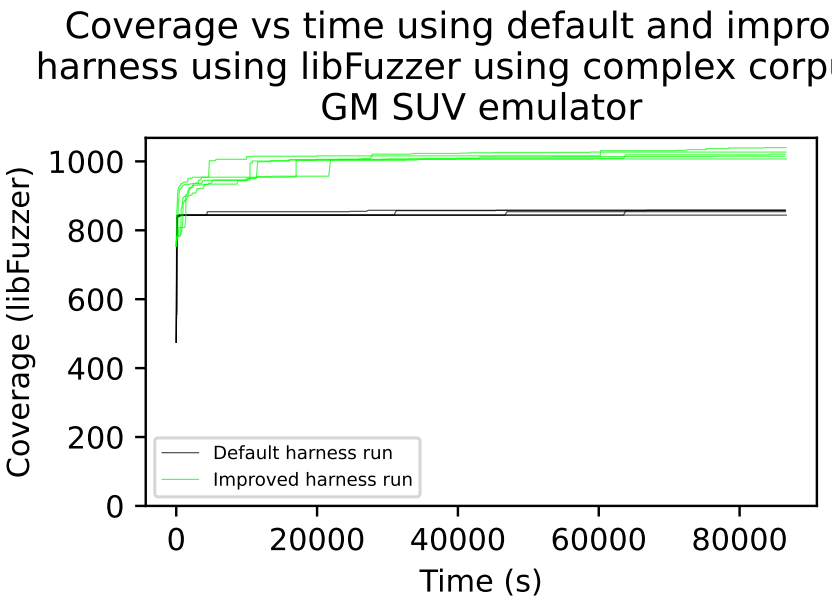

Fig. 22. Coverage vs time using default and improved harness using libFuzzer using complex corpus on GM SUV emulator

8 Discussion

In this section, we discuss and compare the results presented in Section 7. In addition, we state whether our results confirm or not the hypotheses defined in Section 1.2 and discuss their relationship with the research questions from Section 1.1.

The experiment outcomes presented in Section 7.2 showcase that AFL yielded better results than libFuzzer overall regarding coverage. However, when observing default and modified harness versions using AFL-measured coverage, libFuzzer exhibits inferior performance with a modified harness during several runs. In contrast, AFL is continuously surprising with its performance regarding coverage for a modified harness in all runs. When we compare the coverage measured by libFuzzer among the harnesses, we see that in most cases libFuzzer also benefited from a modified harness. Consequently, modifying the harness can increase complexity and impact results. Thus, we confirm Hypothesis 1 in the case of the experiment from Section 7.2.

In addition, the initial corpus used can also negatively affect coverage, depending on the used fuzzer, as we can see when comparing Figures 4 and 9. In this case, AFL has achieved lower coverage when using the improved harness with the complex corpus. However, libFuzzer has achieved higher coverage when fuzzing the improved harness using the complex corpus.

Our findings regarding crash detection in the experiment described in Section 7.2 show that AFL detects some crashes unless they are false positives. Differences in their mechanisms account for these divergent outcomes - AFL has a genetic algorithm-focused approach, whereas libFuzzer places greater emphasis on coverage guidance. Thus, the experiment in 7.2 makes our Hypothesis 2 inconclusive.

When we compare the results from the experiments presented in Section 7.2 and 7.3, the coverage results of the experiment in 7.3 show that the improved harness has better coverage than the default one for libFuzzer and AFL in all five runs. A more complex input seed helps the fuzzer further explore the source code, combined with a harness with higher complexity. Also, the modified harness again shows improved performance with both fuzzers. Finally, we prove our Hypothesis 1 again in the experiment from 7.3.

In the experiment from Section 7.3, libFuzzer fails to detect crashes. Compared to the experiment from Section 7.2, AFL detects fewer crashes using the improved harness than the default one in most runs. We think this happened because of the increased complexity of the input seeds. The fuzzers do more work processing these in the same amount of time. This explication could be testable by running the more complicated harness for longer. Therefore, we again state that our Hypothesis 2 is inconclusive.

The results on coverage increase over time presented in the experiments from Section 7.2, 7.3, and 7.4 show that each fuzzing tool has different coverage increase curves. LibFuzzer excels in achieving coverage fast, but the curve becomes saturated quickly, suggesting it exhausts accessible paths quickly and fuzzing for an extended period is unnecessary in most cases. AFL shows more gradual and consistent coverage growth, especially with the improved harness. Thus, AFL should run longer, especially when fuzzing using more complex harnesses and initial inputs.

Experiments with the Honda emulator described in Section 7.4 show increased code coverage with the improved CAN harness. The improved harness allows for more crash detection—LibFuzzer's non-deterministic behaviour cause the improved harness to have a higher standard deviation. The non-determinism introduces randomness in how fuzzing discovers new code paths to increase coverage. This randomness leads to unpredictability and harder-to-reproduce results. Identical coverage across all runs shows possible coverage ceiling or ineffective corpora. We observe a similar trend with the complex corpus, further solidifying our findings. The improved harness shows the potential to reach higher code coverage. Optimization of corpus selection could improve the fuzzer's effectiveness. Thus, the experiments performed on the Honda emulator are consistent with Hypothesis 2.

Further, the experiments performed on the GM Lyriq and GM SUV show similar trends compared to the results obtained on the Honda emulator. These trends

are similar because all these emulators use Android 11, and the differences among car manufacturers' emulators are insignificant in these cases.

When we compare the results between all the experiments, we observe that coverage in the experiments in Section 7.4 is much lower compared to the other experiments. This difference occurs due to the different implementations of the CAN interface in Android 11. Furthermore, libFuzzer fails to discover crashes in all the experiments. As a result, we confirm our Hypothesis 3.

Introducing existing fuzzers or custom-built fuzzers requires extensive AOSP structure and build system knowledge, as we explain in Section 2.2. Luckily, the build system already integrates libFuzzer and AFL, but they sometimes still require tweaking to compile the fuzzing harnesses or programs under test successfully.

Finally, we confirm our hypotheses 1, 2 and 3 in all of our experiments. Thus, we can answer our sub-research questions 1.1, 1.2, and 1.3, and thus, our main research. Firstly, we answer the research sub-question 1.1 by saying that a more complex harness increases code coverage but not necessarily the number of detected crashes. Secondly, we answer the research sub-research question 1.2 by saying our findings are inconclusive. Thirdly, the different CAN interface implementations among Android Automotive emulators affect the code coverage measured, answering the sub-research question 1.3. Finally, we answer the main research question 1 by stating that fuzzing in an Android Automotive environment performs well, which we can further improve.

9 Limitations

We note some limitations in our research when comparing fuzzing metrics, such as coverage and the number of crashes identified by AFL or libFuzzer. Each framework's performance results in variations for these metrics, so we make comparisons cautiously. Notably, utilizing the "afl-showmap" tool for coverage measurement might add some error in evaluating the results.

Another limitation involves using emulator environments instead of barebone hardware which can affect both the speed and crash detection of a fuzzer like AFL or libFuzzer. So far, no hardware development platform under AOSP presently supports x86(-64). Moreover, the emulation of both the operating system and Android can impact the operation of fuzzing frameworks. For example, coverage tracking might be partially compatible with virtualized environments. These incompatibilities show why we have disabled memory leak detection for libFuzzer for this research.

The hardware we use has limited virtualization features, thus the limited resources for the emulators. This limitation prevents us from using more than one CPU core per emulator. These limitations can impact the speed and efficiency of fuzzing. Fuzzing can benefit from a higher CPU frequency, larger CPU cores, and high SSD I/O speeds.

The Android OS and AOSP environment limits the introduction of new fuzzing frameworks. Introducing new fuzzing engines into the AOSP ecosystem requires extensive modifications in the build system and cross-compilation rules for Android OS. In addition, we need tools such as GDB, Valgrind or cppcheck for extensive investigation of the crashes and memory leaks. These do not come with Android Automotive OS by default and often require cross-compilation, which sometimes becomes technically complex.

Another limitation is not using AFL in the experiments from Section 7.4. We have not been able to use AFL because AOSP developers do not integrate it into the Android 11 build system. Integrating AFL manually requires extensive build system modifications.

Finally, initial input corpora affect the fuzzing process. Thus, our choice of initial inputs might limit our results.

10 Contributions

During our research, we have informed AOSP maintainers or any other related stakeholders about any potential issues identified in AOSP and helped the maintainers.

We have sent updates to the AFL repository that belongs to the AOSP, including new building rules for afl-tmin[7] and afl-gotcpu[8]. We are pleased that the maintainers of the AOSP repositories have integrated these changes for the AFL tools mentioned above.

In addition, we have reported the crash found by AFL to the Android Security team. However, at the time of publishing this thesis, they are still reviewing it.

11 Future Work

Researchers could undertake future work to overcome the limitations mentioned in Section 9. This future work might include exploring different ways of standardizing metrics across fuzzing frameworks and running fuzzing experiments on barebone hardware to see if it affects the speed of fuzzing and the number of crashes detected.

In our Related work section, we described some novel fuzzing frameworks such as Angora, Driller, VUzzer and QSym that we do not use in our research. We would use them but do not because integrating them into the AOSP build system requires a deep understanding. It poses a technical challenge as each fuzzing framework needs specific building rules to implement its binary instrumentation

[7] https://android-review.googlesource.com/c/platform/external/AFLplusplus/+/2475667

[8] https://android-review.googlesource.com/c/platform/external/AFLplusplus/+/2475026

steps. Secondly, some of the fuzzing frameworks need to be more relevant to the experiments done. Finally, some of these fuzzers benefit from experimental setups with better specifications. However, researchers may investigate Android Automotive input interfaces using additional fuzzing frameworks.

As explained in Section 8, our experiments can benefit from fuzzing for an extended time, longer than 24 hours. We recommended repeating our experiments with fuzzing times of at least 48 hours.

Finally, more input interfaces presented in Table 1 should be used for fuzzing experiments. There are more critical interfaces that are of interest regarding fuzzing. Our prioritization of interfaces can be used as a guide in future research.

12 Conclusions

As others describe and show, cyber-physical systems bring new cybersecurity risks to the automotive industry. Researchers have found that systems used for road sign detection and lane departure correction have various threats [2,3,4,5].

The increasing adoption of Android Automotive, an Android OS variant for in-vehicle use, by car manufacturers motivates us to investigate the security aspects of this OS further. Android Automotive interacts with critical vehicle protocols and systems, such as the Controller Area Network (CAN), making any potential vulnerabilities a severe threat.

We use fuzzing techniques to examine the cybersecurity of Android Automotive, focusing on the CAN interface. Our goal is to assess the reliability and safety of the Android Automotive OS and to report any issues discovered to the relevant stakeholders.

The CAN interface is a priority in our research. Our experiments compare different fuzzing engines and emulators. We also contribute to the Android Open Source Project (AOSP) by introducing building rules for two AFL binaries.

In conclusion, our results confirm Hypotheses 1, 2, and 3, and as a result, answering our Sub-research questions 1.1, 1.2 and 1.2. Ultimately, these sub-research questions help us answer the main question 1.

Our experiments demonstrate the improved harness's ability to boost fuzzing performance, thus confirming Hypothesis 1 and answering sub-research question 1.1 by saying that increasing the harness's complexity increases the source code's code coverage or the number of crashes found. AFL's detected potentially false-positive crashes makes Hypothesis 2 inconclusive, and we answer sub-research question 1.2 by stating that our results cannot answer the question.

Moreover, libFuzzer's failure to detect crashes in all experiments further substantiates this hypothesis. Finally, the differences in results across different emulators confirm Hypothesis 3. We answer the sub-research question 1.3 by saying that

the different Android Automotive emulators used influences the source code's coverage or the number or type of crashes.

Finally, we answer the main research question 1 by stating that fuzzing in an Android Automotive environment performs well, which we can improve using complex harnesses, meaningful initial inputs, and the right fuzzing tools.

A Appendix

The interface source code, including the build files, is also available on GitHub[9].

A.1 AFL additional steps

The problem is that we run into "multiple symbols defined" linking errors. We add the following lines in `Android.bp` files of the harnesses when we compile them for AFL usage:

```
ldflags: [
    "-Wl,--allow-multiple-definition",
    "-Wl,--exclude-libs=libclang_rt.fuzzer-x86_64-
       android.a",
],
```

A.2 Modified CAN hardware interface fuzzing harness source code [11]

Source file:

```
/*
 * Copyright (C) 2022 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the
    "License");
 * you may not use this file except in compliance with
    the License.
 * You may obtain a copy of the License at:
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in
    writing, software
 * distributed under the License is distributed on an "
    AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
     express or implied.
 * See the License for the specific language governing
    permissions and
 * limitations under the License.
 *
 */
#include "AutomotiveCanV1_0Fuzzer_Integrated.h"
```

[9] https://github.com/mihaimacarie98/fuzzing_aa_can_interface.git

```
18
19 namespace android::hardware::automotive::can::V1_0::
       implementation::fuzzer {
20
21     constexpr CanController::InterfaceType
           kInterfaceType[] = {
22                 CanController::InterfaceType::VIRTUAL,
23          CanController::InterfaceType::SOCKETCAN,
24          CanController::InterfaceType::SLCAN
25          };
26     constexpr FilterFlag kFilterFlag[] = {FilterFlag::
           DONT_CARE, FilterFlag::SET, FilterFlag::NOT_SET
           };
27     constexpr size_t kInterfaceTypeLength = std::size(
           kInterfaceType);
28     constexpr size_t kFilterFlagLength = std::size(
           kFilterFlag);
29     constexpr size_t kMaxCharacters = 30;
30     constexpr size_t kMaxPayloadBytes = 64;
31     constexpr size_t kMaxFilters = 20;
32     constexpr size_t kMaxSerialNumber = 1000;
33     constexpr size_t kMaxBuses = 10;
34     constexpr size_t kMaxRepeat = 5;
35
36     void CanBusSlcanFuzzer::fuzz() {
37         preUp();
38     }
39
40     void CanBusVirtualFuzzer::fuzz() {
41         preUp();
42     }
43
44     void CanBusNativeFuzzer::fuzz() {
45         preUp();
46     }
47
48     void CanSocketFuzzer::fuzz(FuzzedDataProvider*
           mFuzzedDataProvider) {
49         CanSocket::ReadCallback rdcb =
50          [](const struct canfd_frame& frame, std::chrono
               ::nanoseconds ns) {
51              #pragma unused(frame,ns)
52          };
53
54          CanSocket::ErrorCallback errcb =
```

```
55              [](int errnoVal) {
56              #pragma unused(errnoVal)
57          };
58
59          std::string ifname = mFuzzedDataProvider->
                ConsumeRandomLengthString(
                mFuzzedDataProvider->remaining_bytes() / 2)
                ;
60          auto can_socket = CanSocket::open(ifname, rdcb,
                errcb);
61          if (can_socket && mFuzzedDataProvider->
                remaining_bytes() > 0) {
62              struct canfd_frame frame;
63              std::memset(&frame, 0, sizeof(frame));
64
65              auto data = mFuzzedDataProvider->
                    ConsumeBytes<uint8_t>(std::min(sizeof(
                    frame.data), mFuzzedDataProvider->
                    remaining_bytes()));
66              std::memcpy(frame.data, data.data(), data.
                    size());
67              can_socket->send(frame);
68          }
69      }
70
71      Bus CanFuzzer::makeBus() {
72          ICanController::BusConfig config = {};
73          if (mBusNames.size() > 0 && mLastInterface <
                mBusNames.size()) {
74              config.name = mBusNames[mLastInterface++];
75          } else {
76              config.name = mFuzzedDataProvider->
                    ConsumeRandomLengthString(
                    kMaxCharacters);
77          }
78          config.interfaceId.virtualif({
                mFuzzedDataProvider->
                ConsumeRandomLengthString(kMaxCharacters)})
                ;
79          return Bus(mCanController, config);
80      }
81
82      void CanFuzzer::getSupportedInterfaceTypes() {
83          hidl_vec<CanController::InterfaceType>
                iftypesResult;
```

```
84          mCanController->getSupportedInterfaceTypes(
                hidl_utils::fill(&iftypesResult));
85      }
86
87      hidl_vec<hidl_string> CanFuzzer::getBusNames() {
88          hidl_vec<hidl_string> services = {};
89          if (auto manager = hidl::manager::V1_2::
                IServiceManager::getService(); manager) {
90              manager->listManifestByInterface(ICanBus::
                    descriptor, hidl_utils::fill(&services)
                    );
91          }
92          return services;
93      }
94
95      void CanFuzzer::invokeUpInterface() {
96          const CanController::InterfaceType iftype =
97                  kInterfaceType[mFuzzedDataProvider->
                        ConsumeIntegralInRange<size_t>(
98                            0, kInterfaceTypeLength - 1)];
99          std::string configName;
100
101         if (const bool shouldInvokeValidBus =
                mFuzzedDataProvider->ConsumeBool();
102                 (shouldInvokeValidBus) && (mBusNames.
                        size() > 0)) {
103             const size_t busNameIndex =
104                     mFuzzedDataProvider->
                            ConsumeIntegralInRange<size_t
                            >(0, mBusNames.size() - 1);
105             configName = mBusNames[busNameIndex];
106         } else {
107             configName = mFuzzedDataProvider->
                    ConsumeRandomLengthString(
                    kMaxCharacters);
108         }
109         const std::string ifname = mFuzzedDataProvider
                ->ConsumeRandomLengthString(kMaxCharacters)
                ;
110
111         ICanController::BusConfig config = {.name =
                configName};
112
113         if (iftype == CanController::InterfaceType::
                SOCKETCAN) {
```

```
114             CanController::BusConfig::InterfaceId::
                    Socketcan socketcan = {};
115             if (const bool shouldPassSerialSocket =
                    mFuzzedDataProvider->ConsumeBool();
116                     shouldPassSerialSocket) {
117                 socketcan.serialno(
118                         {mFuzzedDataProvider->
                            ConsumeIntegralInRange<
                            uint32_t>(0,
                            kMaxSerialNumber)});
119             } else {
120                 socketcan.ifname(ifname);
121             }
122             config.interfaceId.socketcan(socketcan);
123         } else if (iftype == CanController::
                InterfaceType::SLCAN) {
124             CanController::BusConfig::InterfaceId::
                    Slcan slcan = {};
125             if (const bool shouldPassSerialSlcan =
                    mFuzzedDataProvider->ConsumeBool();
126                     shouldPassSerialSlcan) {
127                 slcan.serialno(
128                         {mFuzzedDataProvider->
                            ConsumeIntegralInRange<
                            uint32_t>(0,
                            kMaxSerialNumber)});
129             } else {
130                 slcan.ttyname(ifname);
131             }
132             config.interfaceId.slcan(slcan);
133         } else if (iftype == CanController::
                InterfaceType::VIRTUAL) {
134             config.interfaceId.virtualif({ifname});
135         }
136
137         const size_t numInvocations =
138                 mFuzzedDataProvider->
                    ConsumeIntegralInRange<size_t>(0,
                    kMaxRepeat);
139         for (size_t i = 0; i < numInvocations; ++i) {
140             mCanController->upInterface(config);
141         }
142     }
143
144     void CanFuzzer::invokeDownInterface() {
```

```
145         hidl_string configName;
146         if (const bool shouldInvokeValidBus =
                mFuzzedDataProvider->ConsumeBool();
147                 (shouldInvokeValidBus) && (mBusNames.
                        size() > 0)) {
148             const size_t busNameIndex =
149                     mFuzzedDataProvider->
                            ConsumeIntegralInRange<size_t
                            >(0, mBusNames.size() - 1);
150             configName = mBusNames[busNameIndex];
151         } else {
152             configName = mFuzzedDataProvider->
                    ConsumeRandomLengthString(
                    kMaxCharacters);
153         }
154
155         const size_t numInvocations =
156                 mFuzzedDataProvider->
                        ConsumeIntegralInRange<size_t>(0,
                        kMaxRepeat);
157         for (size_t i = 0; i < numInvocations; ++i) {
158             mCanController->downInterface(configName);
159         }
160     }
161
162     void CanFuzzer::invokeController() {
163         getSupportedInterfaceTypes();
164         invokeUpInterface();
165         invokeDownInterface();
166     }
167
168     void CanFuzzer::invokeBus() {
169         const size_t numBuses = mFuzzedDataProvider->
                ConsumeIntegralInRange<size_t>(1, kMaxBuses
                );
170         for (size_t i = 0; i < numBuses; ++i) {
171             if (const bool shouldSendMessage =
                        mFuzzedDataProvider->ConsumeBool();
                    shouldSendMessage) {
172                 auto sendingBus = makeBus();
173                 CanMessage msg = {.id =
                            mFuzzedDataProvider->
                            ConsumeIntegral<uint32_t>()};
174                 uint32_t numPayloadBytes =
```

```
175                     mFuzzedDataProvider->
                            ConsumeIntegralInRange<
                            uint32_t>(0,
                            kMaxPayloadBytes);
176             hidl_vec<uint8_t> payload(
                    numPayloadBytes);
177             for (uint32_t j = 0; j <
                    numPayloadBytes; ++j) {
178                 payload[j] = mFuzzedDataProvider->
                        ConsumeIntegral<uint32_t>();
179             }
180             msg.payload = payload;
181             msg.remoteTransmissionRequest =
                    mFuzzedDataProvider->ConsumeBool();
182             msg.isExtendedId = mFuzzedDataProvider
                    ->ConsumeBool();
183             sendingBus.send(msg);
184         } else {
185             auto listeningBus = makeBus();
186             uint32_t numFilters =
187                     mFuzzedDataProvider->
                            ConsumeIntegralInRange<
                            uint32_t>(1, kMaxFilters);
188             hidl_vec<CanMessageFilter> filterVector
                    (numFilters);
189             for (uint32_t k = 0; k < numFilters; ++
                    k) {
190                 filterVector[k].id =
                        mFuzzedDataProvider->
                        ConsumeIntegral<uint32_t>();
191                 filterVector[k].mask =
                        mFuzzedDataProvider->
                        ConsumeIntegral<uint32_t>();
192                 filterVector[k].rtr =
193                         kFilterFlag[
                                mFuzzedDataProvider->
                                ConsumeIntegralInRange<
                                size_t>(
194                                     0,
                                         kFilterFlagLength
                                          - 1)];
195                 filterVector[k].extendedFormat =
196                         kFilterFlag[
                                mFuzzedDataProvider->
```

```
                                    ConsumeIntegralInRange<
                                    size_t>(
197                                      0,
                                             kFilterFlagLength
                                              - 1)];
198                     filterVector[k].exclude =
                            mFuzzedDataProvider->
                            ConsumeBool();
199                 }
200                 auto listener = listeningBus.listen(
                        filterVector);
201             }
202         }
203     }
204
205     void CanFuzzer::deInit() {
206         mCanController.clear();
207         if (mFuzzedDataProvider) {
208             delete mFuzzedDataProvider;
209         }
210         mBusNames = {};
211     }
212
213     void CanFuzzer::process(const uint8_t *data, size_t
            size) {
214         mFuzzedDataProvider = new FuzzedDataProvider(
                data, size);
215         invokeController();
216         invokeBus();
217         // added direct calls
218         findUsbDevice({mFuzzedDataProvider->
                ConsumeIntegralInRange<uint32_t>(0,
                kMaxSerialNumber)});
219         std::string testString = mFuzzedDataProvider->
                ConsumeRandomLengthString(kMaxCharacters);
220         readSerialNo(testString);
221         getIfaceName(testString);
222         isValidName(testString);
223         // CanBusNativeFuzzer
224         const std::string ifname = mFuzzedDataProvider
                ->ConsumeRandomLengthString(kMaxCharacters)
                ;
225         const uint32_t bitrate = mFuzzedDataProvider->
                ConsumeIntegral<uint32_t>();
226         CanBusNativeFuzzer canFuzzer(ifname, bitrate);
```

```
227         canFuzzer.fuzz();
228         //CanBusSlcan
229         CanBusSlcanFuzzer canFuzzer2(ifname, bitrate);
230         canFuzzer2.fuzz();
231         //CanBusVirtual
232         CanBusVirtualFuzzer canFuzzer3(ifname);
233         canFuzzer3.fuzz();
234         //CanSocket
235         CanSocketFuzzer canFuzzer4;
236         canFuzzer4.fuzz(mFuzzedDataProvider);
237     }
238
239     bool CanFuzzer::init() {
240         mCanController = sp<CanController>::make();
241         if (!mCanController) {
242             return false;
243         }
244         mBusNames = getBusNames();
245         return true;
246     }
247
248     extern "C" int LLVMFuzzerTestOneInput(const uint8_t
          *data, size_t size) {
249         if (size < 1) {
250             return 0;
251         }
252         CanFuzzer canFuzzer;
253         if (canFuzzer.init()) {
254             canFuzzer.process(data, size);
255         }
256         return 0;
257     }
258 } // namespace android::hardware::automotive::can::V1_0
      ::implementation::fuzzer
```

Header file:

```
/*
 * Copyright (C) 2022 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at:
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *
 */
#ifndef __AUTOMOTIVE_CAN_V1_0_FUZZER_INTEGRATED_H__
#define __AUTOMOTIVE_CAN_V1_0_FUZZER_INTEGRATED_H__
#include <CanController.h>
#include <CanBusNative.h>
#include <CanBusVirtual.h>
#include <CanBusSlcan.h>
#include <android/hidl/manager/1.2/IServiceManager.h>
#include <fuzzer/FuzzedDataProvider.h>
#include <hidl-utils/hidl-utils.h>

namespace android::hardware::automotive::can::V1_0::implementation::fuzzer {

    using ::android::sp;
    struct CanSocketFuzzer{
    public:
        void fuzz(FuzzedDataProvider* mFuzzedDataProvider);
    };
    struct CanBusVirtualFuzzer: public CanBusVirtual{
    public:
```

```
36          CanBusVirtualFuzzer(const std::string& ifname)
                : CanBusVirtual(ifname) {};
37          void fuzz();
38      };
39      struct CanBusSlcanFuzzer: public CanBusSlcan{
40      public:
41          CanBusSlcanFuzzer(const std::string& uartName,
                uint32_t bitrate) : CanBusSlcan(uartName,
                bitrate) {};
42          void fuzz();
43          ~CanBusSlcanFuzzer() {   postDown(); }
44      };
45      struct CanBusNativeFuzzer: public CanBusNative{
46      public:
47          CanBusNativeFuzzer(const std::string& ifname,
                uint32_t bitrate) : CanBusNative(ifname,
                bitrate) {};
48          void fuzz();
49      };
50
51      struct CanMessageListener : public can::V1_0::
            ICanMessageListener {
52          DISALLOW_COPY_AND_ASSIGN(CanMessageListener);
53
54          CanMessageListener() {}
55
56          virtual Return<void> onReceive(const can::V1_0
                ::CanMessage& msg) override {
57              std::unique_lock<std::mutex> lock(
                    mMessagesGuard);
58              mMessages.push_back(msg);
59              mMessagesUpdated.notify_one();
60              return {};
61          }
62
63          virtual ~CanMessageListener() {
64              if (mCloseHandle) {
65                  mCloseHandle->close();
66              }
67          }
68
69          void assignCloseHandle(sp<ICloseHandle>
                closeHandle) { mCloseHandle = closeHandle;
                }
70
```

```
71      private:
72          sp<ICloseHandle> mCloseHandle;
73
74          std::mutex mMessagesGuard;
75          std::condition_variable mMessagesUpdated
               GUARDED_BY(mMessagesGuard);
76          std::vector<can::V1_0::CanMessage> mMessages
               GUARDED_BY(mMessagesGuard);
77      };
78
79      struct Bus {
80          DISALLOW_COPY_AND_ASSIGN(Bus);
81
82          Bus(sp<ICanController> controller, const
               ICanController::BusConfig& config)
83                  : mIfname(config.name), mController(
                       controller) {
84              const auto result = controller->upInterface
                   (config);
85              const auto manager = hidl::manager::V1_2::
                   IServiceManager::getService();
86              const auto service = manager->get(ICanBus::
                   descriptor, config.name);
87              mBus = ICanBus::castFrom(service);
88          }
89
90          virtual ~Bus() { reset(); }
91
92          void reset() {
93              mBus.clear();
94              if (mController) {
95                  mController->downInterface(mIfname);
96                  mController.clear();
97              }
98          }
99
100         ICanBus* operator->() const { return mBus.get()
               ; }
101         sp<ICanBus> get() { return mBus; }
102
103         sp<CanMessageListener> listen(const hidl_vec<
               CanMessageFilter>& filter) {
104             sp<CanMessageListener> listener = sp<
                   CanMessageListener>::make();
105
```

```
106             if (!mBus) {
107                 return listener;
108             }
109             Result result;
110             sp<ICloseHandle> closeHandle;
111             mBus->listen(filter, listener, hidl_utils::
                    fill(&result, &closeHandle)).assertOk()
                    ;
112             listener->assignCloseHandle(closeHandle);
113 
114             return listener;
115         }
116 
117         void send(const CanMessage& msg) {
118             if (!mBus) {
119                 return;
120             }
121             mBus->send(msg);
122         }
123 
124     private:
125         const std::string mIfname;
126         sp<ICanController> mController;
127         sp<ICanBus> mBus;
128     };
129 
130     class CanFuzzer {
131     public:
132         ~CanFuzzer() { deInit(); }
133         bool init();
134         void process(const uint8_t* data, size_t size);
135         void deInit();
136 
137     private:
138         Bus makeBus();
139         hidl_vec<hidl_string> getBusNames();
140         void getSupportedInterfaceTypes();
141         void invokeBus();
142         void invokeController();
143         void invokeUpInterface();
144         void invokeDownInterface();
145         FuzzedDataProvider* mFuzzedDataProvider =
                nullptr;
146         sp<CanController> mCanController = nullptr;
147         hidl_vec<hidl_string> mBusNames = {};
```

```
148         unsigned mLastInterface = 0;
149     };
150 }  // namespace android::hardware::automotive::can::
    V1_0::implementation::fuzzer
151
152 #endif  // __AUTOMOTIVE_CAN_V1_0_FUZZER_INTEGRATED_H__
```

Build file:

```
/*
 * Copyright (C) 2022 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at:
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *
 */

package {
    // See: http://go/android-license-faq
    // A large-scale-change added 'default_applicable_licenses' to import
    // all of the 'license_kinds' from "hardware_interfaces_license"
    // to get the below license kinds:
    //   SPDX-license-identifier-Apache-2.0
    default_applicable_licenses: ["hardware_interfaces_license"],
}

cc_fuzz {
    name: "automotiveCanV1.0_fuzzer_integrated",
    vendor: true,
    defaults: ["android.hardware.automotive.can@defaults"],
//    ldflags: [
//        "-fsanitize=address,undefined,fuzzer",
//        "-Wl,--allow-multiple-definition",
```

```
//         "-Wl,--exclude-libs=libclang_rt.fuzzer-x86_64-android.a",
//     ],
//     cflags: [
//         "-fsanitize=address,undefined,fuzzer",
//         "-fsanitize-coverage=trace-pc-guard",
//     ],

    cflags: [
        "-fno-omit-frame-pointer",
    ],

    srcs: [
        "AutomotiveCanV1_0Fuzzer_Integrated.cpp",
        ":automotiveCanV1.0_sources",
    ],
    header_libs: [
        "automotiveCanV1.0_headers",
        "android.hardware.automotive.can@hidl-utils-lib",
    ],
    shared_libs: [
        "android.hardware.automotive.can@1.0",
        "libhidlbase",
    ],
    static_libs: [
        "android.hardware.automotive.can@libnetdevice",
        "android.hardware.automotive@libc++fs",
        "libnl++",
    ],
    fuzz_config: {
        cc: [
            "android-media-fuzzing-reports@google.com",
        ],
        componentid: 533764,
    },
}
```

References

1. GmbH, R.B.: CAN Specification Version 2.0 (1991)
2. New cyber security and software update rules in the automotive industry in 2022 (Jun 2022), https://www.engage.hoganlovells.com/knowledgeservices/viewContent.action?key=Ec8teaJ9VapgpeSCnunnmsxgHJMKLFEppVpbbVX%2B3OXcP3PYxlq7sZUjdbSm5FIetvAtgf1eVU8%3D&nav=FRbANEucS95NMLRN47z%2BeeOgEFCt8EGQ0qFfoEM4UR4%3D&emailtofriendview=true&freeviewlink=true, [Online; accessed 14. Jun. 2022]
3. Contributors to Wikimedia projects: Advanced driver-assistance system - Wikipedia (Jun 2022), https://en.wikipedia.org/w/index.php?title=Advanced_driver-assistance_system&oldid=1095051291, [Online; accessed 26. Jun. 2022]
4. 3 noteworthy automotive system trends for 2022 and beyond - Tuxera (Apr 2022), https://www.tuxera.com/blog/3-automotive-system-trends-2022-beyond, [Online; accessed 26. Jun. 2022]
5. Kim, K., Kim, J.S., Jeong, S., Park, J.H., Kim, H.K.: Cybersecurity for autonomous vehicles: Review of attacks and defense. Computers & Security **103**, 102150 (2021). https://doi.org/https://doi.org/10.1016/j.cose.2020.102150, https://www.sciencedirect.com/science/article/pii/S0167404820304235
6. Protalinski, E.: Google opens android automotive to app developers (May 2019), https://venturebeat.com/business/google-opens-android-automotive-to-app-developers/, accessed: 2023-06-19
7. Building infotainment system powered by android automotive os — infopulse, https://www.infopulse.com/blog/how-to-build-a-customer-tailored-infotainment-system-powered-by-android-automotive-os, (Accessed on 06/23/2022)
8. Project, A.O.S.: Initializing a build environment (2023), https://source.android.com/docs/setup/start/initializing, accessed: 2023-06-04
9. What is Android Automotive? | Android Open Source Project (Jun 2022), https://source.android.com/devices/automotive/start/what_automotive, [Online; accessed 11. Jun. 2022]
10. Pese, M., Shin, K., Bruner, J., Chu, A.: Security analysis of android automotive. SAE International Journal of Advances and Current Practices in Mobility **2**(4), 2337–2346 (apr 2020). https://doi.org/https://doi.org/10.4271/2020-01-1295, https://doi.org/10.4271/2020-01-1295
11. Project, A.O.S.: Automotivecanv1_0fuzzer (2023), https://cs.android.com/android/platform/superproject/+/master:hardware/interfaces/automotive/can/1.0/default/tests/fuzzer/, accessed: 2023-07-02
12. ISO 26262: The ISO Standard for Functional Safety (Mar 2022), https://securityboulevard.com/2022/03/iso-26262-the-iso-standard-for-functional-safety, [Online; accessed 14. Jun. 2022]
13. UN Regulation No. 155 - Cyber security and cyber security management system | UNECE (Jun 2022), https://unece.org/transport/documents/2021/03/standards/un-regulation-no-155-cyber-security-and-cyber-security, [Online; accessed 14. Jun. 2022]
14. UN Regulation No. 156 - Software update and software update management system | UNECE (Jun 2022), https://unece.org/transport/documents/2021/03/standards/un-regulation-no-156-software-update-and-software-update, [Online; accessed 14. Jun. 2022]

15. Moiz, A., Alalfi, M.H.: An approach for the identification of information leakage in automotive infotainment systems. In: 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 110–114 (2020). https://doi.org/10.1109/SCAM51674.2020.00017
16. What is code coverage and how to measure it? code coverage benefits. https://www.codegrip.tech/productivity/everything-you-need-to-know-about-code-coverage, (Accessed on 07/08/2022)
17. Dechand, S.: The Magic Behind Feedback-Based Fuzzing. Code Intelligence (Jun 2022), https://www.code-intelligence.com/blog/the-magic-behind-feedback-based-fuzzing
18. Contributors to Wikimedia projects: Instrumentation (computer programming) - Wikipedia (Dec 2020), https://en.wikipedia.org/w/index.php?title=Instrumentation_(computer_programming)&oldid=997174211, [Online; accessed 23. Jun. 2022]
19. Lee, H., Choi, K., Chung, K., Kim, J., Yim, K.: Fuzzing can packets into automobiles. In: 2015 IEEE 29th International Conference on Advanced Information Networking and Applications. pp. 817–821 (2015). https://doi.org/10.1109/AINA.2015.274
20. Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: AFL++ : Combining incremental steps of fuzzing research. In: 14th USENIX Workshop on Offensive Technologies (WOOT 20). USENIX Association (Aug 2020), https://www.usenix.org/conference/woot20/presentation/fioraldi
21. Zalewski, M.: American fuzzy lop - whitepaper (2016), https://lcamtuf.coredump.cx/afl/technical_details.txt, [Online; accessed 24. Jun. 2022]
22. libFuzzer – a library for coverage-guided fuzz testing. — LLVM 15.0.0git documentation (Jun 2022), https://llvm.org/docs/LibFuzzer.html, [Online; accessed 24. Jun. 2022]
23. Fuzzing: Common Tools and Techniques — coalfire.com. https://www.coalfire.com/the-coalfire-blog/fuzzing-common-tools-and-techniques, [Accessed 14-Jun-2022]
24. Zeller, A., Gopinath, R., Böhme, M., Fraser, G., Holler, C.: Greybox fuzzing. In: The Fuzzing Book. CISPA Helmholtz Center for Information Security (2022), https://www.fuzzingbook.org/html/GreyboxFuzzer.html, retrieved 2022-05-17 18:23:54+02:00
25. Fuzzing techniques - the generator menace - coders kitchen. https://www.coderskitchen.com/fuzzing-techniques/, (Accessed on 06/22/2022)
26. Godefroid, P., Levin, M.Y., Molnar, D.: Sage: Whitebox fuzzing for security testing: Sage has had a remarkable impact at microsoft. Queue **10**(1), 20–27 (jan 2012). https://doi.org/10.1145/2090147.2094081, https://doi.org/10.1145/2090147.2094081
27. Aki Helin / radamsa · GitLab (Jun 2022), https://gitlab.com/akihe/radamsa, [Online; accessed 24. Jun. 2022]
28. RootUp: BFuzz (Jun 2022), https://github.com/RootUp/BFuzz, [Online; accessed 24. Jun. 2022]
29. ClusterFuzz (Jun 2022), https://google.github.io/clusterfuzz, [Online; accessed 30. Jun. 2022]
30. Coverage guided vs blackbox fuzzing (Jun 2022), https://google.github.io/clusterfuzz/reference/coverage-guided-vs-blackbox, [Online; accessed 24. Jun. 2022]
31. Blackbox fuzzing (Jun 2022), https://google.github.io/clusterfuzz/setting-up-fuzzing/blackbox-fuzzing, [Online; accessed 24. Jun. 2022]

32. american fuzzy lop (Jun 2022), https://lcamtuf.coredump.cx/afl, [Online; accessed 24. Jun. 2022]
33. Peach Fuzzer (Mar 2021), https://peachtech.gitlab.io/peach-fuzzer-community, [Online; accessed 24. Jun. 2022]
34. k0retux: fuddly (Jun 2022), https://github.com/k0retux/fuddly, [Online; accessed 24. Jun. 2022]
35. Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H.: VUzzer : Application - aware Evolutionary Fuzzing, p. 14. NDSS'17, NDSS'17 (2017). https://doi.org/10.14722/ndss.2017.23404
36. TechTarget Contributor: fuzz testing (fuzzing). SearchSecurity (Mar 2010), https://www.techtarget.com/searchsecurity/definition/fuzz-testing
37. Perl, H.: What Bugs Can You Find With Fuzzing? Code Intelligence (Jun 2022), https://www.code-intelligence.com/blog/what-bugs-can-you-find-with-fuzzing
38. Automotive | Android Open Source Project (Jun 2022), https://source.android.com/devices/automotive, [Online; accessed 14. Jun. 2022]
39. Vehicle properties — android open source project. https://source.android.com/devices/automotive/vhal/properties, (Accessed on 06/23/2022)
40. Road vehicles – controller area network (can) – part 1: Data link layer and physical signalling (2015)
41. Nishimura, R., Kurachi, R., Ito, K., Miyasaka, T., Yamamoto, M., Mishima, M.: Implementation of the can-fd protocol in the fuzzing tool bestorm. In: 2016 IEEE International Conference on Vehicular Electronics and Safety (ICVES). pp. 1–6 (2016). https://doi.org/10.1109/ICVES.2016.7548161
42. Contributors to Wikimedia projects: ISO 26262 - Wikipedia (Apr 2022), https://en.wikipedia.org/w/index.php?title=ISO_26262&oldid=1084731619, [Online; accessed 14. Jun. 2022]
43. What Is ISO 26262? Overview and ASIL | Perforce Software (Jun 2022), https://www.perforce.com/blog/qac/what-is-iso-26262, [Online; accessed 14. Jun. 2022]
44. UN Regulation No 155 & how to comply? What you need to know (Jun 2022), https://www.cyres-consulting.com/un-regulation-no-155-requirements-what-you-need-to-know/#What_are_the_UN_R155_requirements, [Online; accessed 14. Jun. 2022]
45. White Paper: UNECE Cybersecurity Regulation (R155) | Secura (Jun 2022), https://www.secura.com/nl/whitepapers/unece-r155, [Online; accessed 14. Jun. 2022]
46. UNECE Vehicle Regulation for Cyber Security & Software Updates (Nov 2021), https://conti-engineering.com/unece-vehicle-regulation-for-cyber-security-software-updates, [Online; accessed 14. Jun. 2022]
47. Helix qac for c and c++ — perforce. https://www.perforce.com/products/helix-qac, (Accessed on 07/08/2022)
48. Mayhem for code — forallsecure. https://forallsecure.com/mayhem-for-code, (Accessed on 07/08/2022)
49. Rouf, I., Miller, R., Mustafa, H., Taylor, T., Oh, S., Xu, W., Gruteser, M., Trappe, W., Seskar, I.: Security and privacy vulnerabilities of In-Car wireless networks: A tire pressure monitoring system case study. In: 19th USENIX Security Symposium (USENIX Security 10). USENIX Association, Washington, DC (Aug 2010), https://www.usenix.org/conference/usenixsecurity10/security-and-privacy-vulnerabilities-car-wireless-networks-tire-pressure
50. Emura, K., Hayashi, T., Moriai, S.: Toward securing tire pressure monitoring systems: A case of present-based implementation. In: 2016 International Symposium on Information Theory and Its Applications (ISITA). pp. 403–407 (2016)

51. Kulandaivel, S., Jain, S., Guajardo, J., Sekar, V.: Cannon: Reliable and stealthy remote shutdown attacks via unaltered automotive microcontrollers. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 195–210 (2021). https://doi.org/10.1109/SP40001.2021.00122
52. Antonioli, D., Payer, M.: On the insecurity of vehicles against protocol-level bluetooth threats. In: IEEE (ed.) WOOT 2022, 17th Workshop On Offensive Technologies, co-located with IEEE S&P, 26 May 2022, San Francisco, CA, USA. San Francisco (2022), © 2022 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
53. Antonioli, D., Tippenhauer, N.O., Rasmussen, K.: Bias: Bluetooth impersonation attacks. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P) (May 2020)
54. Antonioli, D., Tippenhauer, N.O., Rasmussen, K.B.: The KNOB is broken: Exploiting low entropy in the encryption key negotiation of bluetooth BR/EDR. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 1047–1061. USENIX Association, Santa Clara, CA (Aug 2019), https://www.usenix.org/conference/usenixsecurity19/presentation/antonioli
55. Sun, J., Cao, Y., Chen, Q.A., Mao, Z.M.: Towards robust lidar-based perception in autonomous driving: General black-box adversarial sensor attack and countermeasures (2020). https://doi.org/10.48550/ARXIV.2006.16974, https://arxiv.org/abs/2006.16974
56. Chen, P., Chen, H.: Angora: Efficient fuzzing by principled search (2018). https://doi.org/10.48550/ARXIV.1803.01307, https://arxiv.org/abs/1803.01307
57. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: NDSS 2016 (01 2016). https://doi.org/10.14722/ndss.2016.23368
58. Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T.: QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 745–761. USENIX Association, Baltimore, MD (Aug 2018), https://www.usenix.org/conference/usenixsecurity18/presentation/yun
59. Fujikura, T., Kurachi, R., Oka, D.: Shift left: Fuzzing earlier in the automotive software development lifecycle using hil systems. In: escar Europe 2018 (11 2018)
60. Radu, A.I., Garcia, F.D.: Grey-box analysis and fuzzing of automotive electronic components via control-flow graph extraction. In: Computer Science in Cars Symposium. CSCS '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3385958.3430480, https://doi.org/10.1145/3385958.3430480
61. Fowler, D.S., Bryans, J., Shaikh, S.A., Wooderson, P.: Fuzz testing for automotive cyber-security. In: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W). pp. 239–246 (2018). https://doi.org/10.1109/DSN-W.2018.00070
62. Werquin, T., Hubrechtsen, M., Thangarajan, A., Piessens, F., Muehlberg, J.: Automated fuzzing of automotive control units (02 2021)
63. Dynamic application security testing software — beyond security. https://www.beyondsecurity.com/solutions/bestorm-dynamic-application-security-testing.html, (Accessed on 07/08/2022)

64. Defensics fuzz testing tool & services — synopsys. https://www.synopsys.com/software-integrity/security-testing/fuzz-testing.html, (Accessed on 07/08/2022)
65. boofuzz: Network protocol fuzzing for humans — boofuzz 0.4.1 documentation. https://boofuzz.readthedocs.io/en/stable/, (Accessed on 07/08/2022)
66. Huracan: Automotive Security Testing Tool - Riscure (Dec 2021), https://www.riscure.com/security-tools/huracan-automotive-security-tools, [Online; accessed 15. Jun. 2022]
67. Fuzzing | Block Harbor Cybersecurity | Vulnerability Bruteforcing (Nov 2021), https://blockharbor.io/services/fuzzing, [Online; accessed 15. Jun. 2022]
68. Project, A.O.S.: Can interface source code (2023), https://cs.android.com/android/platform/superproject/+/master:hardware/interfaces/automotive/can/1.0/default/tests/fuzzer/, accessed: 2023-07-02