Grey-box Analysis and Fuzzing of Automotive Electronic Components via Control-Flow Graph Extraction

Andreea-Ina Radu
Flavio D. Garcia
a.i.radu@cs.bham.ac.uk
f.garcia@cs.bham.ac.uk
University of Birmingham
Birmingham, United Kingdom

ABSTRACT

Electronic Control Units are embedded systems which control the functionality of a modern vehicle. The growing number of Electronic Control Units in a vehicle, together with their increasing complexity, prompts the need for automated tools to test their security.

To this end, we present EffCAN, a tool for ECU firmware fuzzing via Controller Area Network. EffCAN operates on the Control Flow Graph, which we extract from the firmware. The Control Flow Graph is a platform independent representation, which allows us to abstract from the often obscure underlying architecture. The Control Flow Graph is annotated with information about static data comparisons that affect the control flow of the firmware. This information is used to create initial seeds for the fuzzer. It is also used to adapt the input messages in order to cover hard to reach execution paths. We have evaluated EffCAN on three Electronic Control Units, from different manufacturers. The fuzzer was able to crash two of the units. To our knowledge, this is the first approach that uses static analysis to guide the fuzzing of automotive Electronic Control Units.

CCS CONCEPTS

• **Computer systems organization** → **Firmware**; • **Security and privacy** → **Systems security**.

KEYWORDS

automotive, electronic control unit, fuzzing

ACM Reference Format:
Andreea-Ina Radu and Flavio D. Garcia. 2020. Grey-box Analysis and Fuzzing of Automotive Electronic Components via Control-Flow Graph Extraction. In *Computer Science in Cars Symposium (CSCS '20), December 2, 2020, Feldkirchen, Germany.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3385958.3430480

1 INTRODUCTION

Until recently, the in-vehicle network was considered a safe, trusted environment. Resistance against inside malicious adversaries was not considered. Vehicles have not benefited from the scrutiny that many of the computerised systems surrounding us have, until recently. Nowadays, a vehicle has upwards of 70 Electronic Control Units (ECUs) and, the manufacturer outsources the design of the hardware and development of the firmware to first tier suppliers. First tier suppliers may, in turn, outsource tasks to other companies. The many stakeholders and complexities of the supply chain make tracking of responsibility difficult.

Often, when firmware is outsourced, manufacturers receive only the firmware image, which they upload to the hardware. The source code is not shared with the manufacturer, which means no code auditing can take place, and component testing is limited to the requirements specifications. Furthermore, code reuse is a common practice in software development, but it has been shown to enable vulnerabilities to infiltrate into software [Hanna et al. 2012; Pham et al. 2010; Xia et al. 2014]. Creating a tool to test the ECUs using a gray box approach will allow manufacturers to gain some security guarantees about the outsourced firmware.

Challenges. The hardware and architectures of ECUs are hugely diverse. Most ECUs have bare-metal firmware, (i.e. they do not have an operating system), and directly interface with low-level hardware and peripherals. Additionally, the Microcontroller Units (MCUs) used in ECUs are automotive-specific. They have extra functionality and registers (e.g. for Controller Area Network (CAN) communication), when compared to their generic counterparts.

ECUs are built on a wide range of chips and architectures, which makes creating an automated, 'universal', solution for analysing their firmware difficult, and it is hard to scale up the processes. Binary Independent Languages (BILs) are designed to fill this gap (e.g. BAP[1], VEX[2]), by providing an abstraction layer from the underlying instruction set, and using an Intermediate Representation (IR) to describe the operations performed by the MCU. However, existing BILs support only a limited number of mainstream architectures. BAP [Brumley et al. 2011] supports ARM, x86, x86−64, PowerPC (PPC), and MIPS. VEX requires an operating system, not providing support for bare-metal firmware [Developers 2017]. This means the technique of lifting the ECU firmware to an IR would be too cumbersome to contemplate. Another possible approach to

[1]`https://github.com/BinaryAnalysisPlatform/bap`
[2]`https://docs.angr.io/advanced-topics/ir`

analyse embedded firmware is by running it in a simulated environment, but emulators which support automotive MCUs do not yet exist, therefore profiles for these would need to be developed. In [Gustafson et al. 2019], Gustafson et. al. explain that the abundance of incompatible embedded MCUs and the vast collection of peripherals make emulators with extensive embedded device support unattainable. They also highlight that analysing embedded device firmware entails a considerable manual and time-consuming effort, due to the absence of standards for components and protocols.

Analysing bare-metal firmware requires in-depth knowledge about the underlying hardware, and documentation is an important resource. Obtaining the user manuals for the automotive-specific chips proved to be a difficult task, as they were often not available.

In terms of tools, the only feasible choice is Interactive DisAssembler (IDA), a state-of-the-art disassembler. IDA is the only disassembler which had some form of support for all the architectures and MCUs analysed. However, IDA does not have profiles for the automotive versions of the MCUs, and additional annotation with the extra functionality is required.

When loading bare-metal firmware into IDA, further information needs to be supplied, such as reset vectors. These are considered the *entry points* of the firmware. IDA is able to disassemble by itself only minor parts (if any) of the firmware. To achieve maximal code coverage, we explore a number of architecture specific solutions in Section 3. Extracting an accurate Control Flow Graph (CFG) from a binary and ensuring correct function detection is an ongoing open problem [Andriesse et al. 2017; Bao et al. 2014; Bruschi et al. 2006; Kinder and Veith 2008; Nguyen et al. 2013; Shin et al. 2015], especially with respect to resolving indirect branch instructions. Therefore, some manual guidance and clean-up is still required, but the time needed to bring the disassembled code to a satisfying level is considerably reduced. By manual guidance and clean-up, we refer to the user going through the disassembly and making *informed decisions* on whether the instructions recovered and functions created are correct or not.

As Muench et. al. [Muench et al. 2018] point out in their research, fuzz-testing embedded devices differs from fuzzing desktop systems, due to the limited IO and computing power. They argue that *silent memory corruptions* are much more frequent on embedded devices and they pose a challenge when trying to detect *what* the exact state of the device was when it crashed. Unexpected behaviours could be immediate and observable, delayed, or malfunctions (where the device memory is corrupted and computations are incorrect, but there is no crash). Their arguments are entirely transferable to ECU fuzzing, and identifying such behaviours was the main challenge for our fuzzer. We had to define what it means for an ECU to *crash*. One way of determining crashes is observing the output of the program under test. However, when working with ECUs via CAN communication, the output from the devices is extremely limited. The CAN bus is a broadcast network, where ECUs have pre-established message identifiers (IDs) they use to send their data on. The only bi-directional communication is for Unified Diagnostics Services, which is used for diagnostic purposes and do not appear in normal vehicle operation. Therefore, there is no real method of establishing the internal state of the ECU, or if any error occurred. Also, obtaining execution traces from a component is a difficult, if not impossible, task as ECUs often times lack such debugging support.

Furthermore, when fuzzing CAN communication, the CAN IDs the component uses in its communication must be taken into account. In a scenario where the ECU is nothing more than a black box, the IDs become part of the search space, significantly increasing the time needed for fuzzing. The communication matrix of a vehicle, or of an ECU, describes which signals are sent and received by which ECU, and on which IDs. Such knowledge would be very useful for the fuzzing process: it would allow us to target only IDs the ECU listens to. However, a communication matrix is a well kept industry secret and not readily available.

Contribution. This paper proposes a fuzzer for ECUs which uses information from the ECU firmware in guiding its fuzz testing process. The purpose of the tool is to automatically test ECUs for vulnerabilities or, more generally, for bugs which would lead to system crashes.

We propose a method for extracting the CFG of the firmware and annotating it with information about static data used in deciding the control flow of the program. The method also involves semi-automated instruction recovery and function detection for ECU firmware, a challenging task because ECUs are built on a diverse set of architectures and hardware.

We present a fuzzer, EffCAN, which communicates with an ECU via CAN. The fuzzer uses the static data information from the extracted CFG in creating initial input seeds, and prioritises hard to reach paths in the input transformation procedure. To the best of our knowledge, this work presents the first fuzzer which does not solely rely on randomness in forming CAN messages.

We have evaluated the fuzzer on three ECUs, with positive results on two of them.

2 RELATED WORK

In this section we review relevant literature in connection to firmware analysis and to fuzzing.

Firmware analysis. The vast amount of program analysis research has been concerned mostly with binaries that run within an operating system. However, ECUs have firmware which presents itself as a single, monolithic binary. The firmware binary contains all the functionality of the device and the interaction with the hardware is done in a direct manner. This type of firmware is highly tailored to the underlying hardware it runs on and requires strong knowledge about the device architecture and instruction set used by the MCU.

In the context of automotive security, remote keyless entry systems have received a lot of attention [Garcia et al. 2016; Hicks et al. 2018; Verdult et al. 2015]. Assessing their security usually involves reverse engineering a (proprietary) cipher from the firmware of the immobiliser or ECU and finding weaknesses. However, the reverse engineering process is never fully described, as the cryptographic algorithms are the main scope of the research. Similarly, ciphers used in authenticating to diagnostic protocols running on ECUs are reverse engineered in [den Herrewegen and Garcia 2018], but there is no description of the process.

The only instance of detailed ECU firmware analysis comes from Miller and Valasek in [Miller and Valasek 2015]. They reverse engineered the firmware from a V850 chip which was responsible for dealing with the CAN communication on the ECU they were investigating. They explain that the procedure took them *'several weeks'*, highlighting the extensive effort required. Their ultimate goal was to modify the firmware to accept commands via Serial Peripheral Interface and send CAN messages based on these. Interestingly, the authors mention they had disassembled parts of the firmware incorrectly, treating areas which were supposed to be data as code, and this lead to further delays and confusion. From their research we can learn important information, such as the fragility of the disassembly process and the value of understanding the hardware specification and architecture, information which we applied in Section 3.

Fuzzing. Most fuzzers proposed in the literature [Rawat et al. 2017; Stephens et al. 2016; Zalewski 2014], are targeted at operating and file systems, or protocols, and are not suitable for fuzzing ECUs. They require the ability to execute (part of) the program being tested, to emulate it, or require information about what execution path was triggered by the given input. As previously mentioned, no viable emulators for automotive MCUs exist, and obtaining execution traces from the hardware is often impossible.

A few instances of applying fuzzing to test the security of the automotive components and CAN bus exist, but they all rely on randomly choosing the bytes of a CAN messages.

Lee et. al. [Lee et al. 2015] demonstrated they could sniff the packets from a vehicle, through an On-Board Diagnostics (OBD-II) Bluetooth dongle, and use the knowledge acquired about which CAN IDs are in use. They then fuzzed each byte of the 8 bytes of the possible payload by choosing a random value, and setting the rest to zero. They observed changes in the Instrument Panel Cluster (IPC) signals or in physical components in the vehicle (e.g. lights).

Bayer and Ptok [Bayer and Ptok 2015] present an Unified Diagnostics Services (UDS) fuzzer, which is able to create messages with sequence numbers, as expected by the protocol. They claim the fuzzer is block-based, in that it understands and is aware of what specific fields within the message mean. It can therefore automatically produce correct values for these (e.g. for a checksum field). No other information is given about the strategies the tool uses.

Fowler et. al. [Fowler et al. 2017] argue that the existing design process for ECUs should be extended with automated fuzz-testing, informed by the connections and data from of the in-vehicle network, and describe such a fuzzer in [Fowler et al. 2018]. Their tool fuzzes CAN packets payloads and IDs, and uses the bit flip strategy, randomising CAN payloads. They evaluated the tool against a simulator, an actual ECU (an IPC), the fuzzer was able to trigger odd behaviour, such as negative Rotations Per Minute, activating warning lights and sounds, and displaying a *crash* message. The latter behaviour persisted through power cycles of the IPC, and the researchers decided to limit full vehicle testing to a small subset of CAN IDs.

Patki, Gothindikar and Mane [Patki et al. 2018] present another UDS fuzzer, which uses valid UDS messages and 'mutates' the payload in order to create invalid messages, which are then sent to the ECU being tested. They create invalid messages by using invalid values in the Data Length Code (DLC) field, invalid values for the service sub-functions or invalid inputs in which all bytes are `0x00` or `0xFF`. The authors report that some services do not respond according to the specification of the UDS standard [ISO 2013].

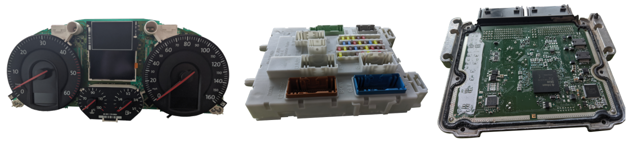
Figure 1: From left to right, the ECUs we worked with are: VW IPC, Ford BCM, Ford ECM.

3 EXTRACTING THE CONTROL FLOW GRAPH FROM ECU FIRMWARE

Program analysis is widely used for understanding firmware functionality, in the absence of the source code. Disassembling is, most commonly, the first step in the analysis, involving a disassembler to translate machine code from a compiled binary to a low-level language, assembly. Due to the obscure architectures ECUs tend to use, even IDA does not handle these binaries gracefully. Therefore, analysing ECUs firmware can be a laborious and tedious, manual task. Even simply loading the firmware correctly into IDA requires manually creating the memory layout, identifying the entry point(s) and mapping of registers for automotive-specific functionality. Additionally, the task of distinguishing code from data sections is left to the user, thus requiring strong knowledge of the device architecture and being able to discern between genuine code and wrongly interpreted code.

Firmware acquisition. We analysed the firmware of three ECUs. We obtained the firmware from the ECUs flash memory by using automotive programmers [usp [n.d.]].

Tools. We used IDA for disassembling the firmware, automated by developing our own IDAPython scripts. We used the `igraph` Python module for storing and working with the CFG, as it allows labelling of vertices and edges, and has good support for efficiently saving the graph to storage.

The ECUs we worked with are shown in Figure 1:

- Volkswagen Passat IPC — ARM architecture (ARM CDC 3297G-C MCU);
- Ford Fiesta Body Control Module (BCM) — PPC architecture (SPC560B);
- Ford Kuga Engine Control Module (ECM) — Infineon TriCore architecture (SAK-TC1793F MCU).

3.1 Register Documentation

One of the first issues tackled is the lack of automotive-specific knowledge of IDA about the MCUs. For example, for the ARM ECU (ARM CDC 3297G-C MCU), one can choose the ARM processor, and specify the start address and size for the Random Access Memory (RAM) and Read Only Memory (ROM) sections, as well as the

```
{
  "can_comm_objects": {
    "label" : "CO",
    "base_addr" : "0xF80000",
    "include_label": "1",
    "iterations" : "5",
    "reg_size" : "1",
    "size": "0x10",
    "payload": {
      "CTRL": "0x00",
      "ID28.21": "0x01",
      "ID20.13": "0x02",
      "ID12.05": "0x03",
      "ID04.00_CTRL": "0x04",
      "DLC_CTRL": "0x05",
      "Data": {
        "offset": "0x06",
        "iterations": "8"
      },
      "timestamp_low": "0x0d",
      "timestamp_high": "0x0e"
    }
  }
}
```

```
IO:F80000 CO0_CTRL           % 1
IO:F80001 CO0_ID28.21        % 1
IO:F80002 CO0_ID20.13        % 1
IO:F80003 CO0_ID12.05        % 1
IO:F80004 CO0_ID04.00_CTRL   % 1
IO:F80005 CO0_DLC_CTRL       % 1
IO:F80006 CO0_Data0          % 1
IO:F80007 CO0_Data1          % 1
IO:F80008 CO0_Data2          % 1
IO:F80009 CO0_Data3          % 1
IO:F8000A CO0_Data4          % 1
IO:F8000B CO0_Data5          % 1
IO:F8000C CO0_Data6          % 1
IO:F8000D CO0_Data7          % 1
IO:F8000E CO0_timestamp_low  % 1
IO:F8000F CO0_timestamp_high % 1
```

Figure 2: Example describing registers as JavaScript Object Notation (JSON), for the ARM CDC 3297G MCU, on the left; IDA excerpt, after applying the registers mapping for the first CAN Communication Object, on the right. The %1 value for each register denotes the size (1 byte).

offset at which the binary should be loaded. IDA presents the user with a warning that it cannot analyse the binary automatically, and displays the byte values of the firmware. The MCU datasheet provides information of the memory layout, what addresses do the RAM, ROM, Input/Output (IO) segments have, etc. This information needs to be manually transferred into IDA, in order to create a correct MCU profile.

In order to help document the registers, we described the registers in JSON format and developed a script that annotates annotates the names, creates the appropriate sizes, and adds comments. Figure 2 shows an excerpt from applying the mapping to the IDA database. If the keyword `"payload"` exists, the object is considered a *group* of registers. The `"base_addr"` keyword contains the address from which the group starts, and each register will contain an `"offset"`, which is added to the base address to obtain the register's address. If the `"payload"` keyword does not exist, the structure is considered one single register. The size of the registers from the group can be specified with the `"reg_size"` property, but it can also be declared for one specific register. The `"code"` keyword means that instead of converting the address to a byte/word/dword, the script will try to create an instruction. Setting the `"include_label"` property to `True` (1) will prepend the `"label"` to the register name. In some cases, objects are mapped to registers, such as in the case of the CAN Communication Objects. Multiple such objects are defined in the CAN-RAM area of the IO segment, and contain CAN messages either received or scheduled to be broadcast. The size of the object can be defined by using the `"size"` keyword, and the number of objects to be mapped can be specified through the `"iterations"` property.

Mapping interesting registers, such as CAN, CAN message buffers, Interrupt Source Nodes (ISNs) or ports, helps improve the readability of the code when manually inspecting it. By documenting them in JSON format, together with the mapping script, the aim is to have a structured and extensible way of dealing with the lack of support and of easily adding the missing information to IDA. The method can be improved by combining the script with a PDF scraper, which could automatically generate the JSON file, as many datasheets have tables which specify the base address, offset and name of each register. IDA allows additional annotation of registers, interrupts and ports through its own `.cfg` files. However, in the `.cfg` files, each register needs to be defined individually. In the case of the example above, for CAN Communication Objects, each register would have to be defined 5 times (as the number of iterations is 5). Our method is much more flexible, and more compact. It can, in fact, be used to generate IDA `.cfg` files, if so desired, in a much more efficient way.

3.2 Disassembling Electronic Control Unit Firmware

As IDA does not recognise the entry point of the firmware binary, guidance needs to be provided. Firstly, the reset vector is given as entry point. Based on the datasheets, the reset vector is, generally, at the first address in ROM. This will reveal the main loop of the firmware. Other registers can be considered entry points, such as the ISNs, Software Interrupt vector, CAN Interrupt Index Register. This applies to all MCUs in the test set and can be used across architectures. However, using these entry points does not guarantee that the whole program will be covered. Indirect jumps or jump tables may not be recognised by the disassembler, and therefore need manual intervention. Also, there may be areas of unreachable code, possible leftovers from testing the firmware. The remainder of this section explains the methods tested on each architecture. The effort of establishing these methods is a one-off cost, as they can be used on all ECUs with MCUs of the architecture.

Disassembling ARM. With respect to our ARM ECU, once we have loaded the firmware, other architecture options can be set, such as the version of the ARM instruction set, whether the board is a Variable Floating Point co-processor, or it uses the advanced single instruction multiple data extension (mainly used in signal processing for media applications). All these pieces of information require a strong understanding of the hardware we are working with. The datasheet of the CDC 3297G chip specifies it uses ARM7TDMI MCU, which leads us to believe it uses the ARMv4T instruction set (Thumb enabled)[3].

One of the heuristics tested was *forcefully creating functions* at each address in the ROM segment of the firmware. As IDA cannot distinguish between `.code` and `.data` areas in the ROM, there will be areas where data is stored, but is wrongly disassembled as code. In order to overcome this, the functions that were successfully created are logged into a file, and a clean database of the firmware is

[3]ARM7TDMI first appears in MCUs from the 1990s, which used the ARMv3 instruction set. However, it also appears in MCUs from the 2000s and later, which use the ARMv4 instruction set. No further information is given in the datasheets, but the brief document for the CDC 32xxG MCU family, detailing their suitability as car dashboard controllers, is dated 2000 [bri 2000] and therefore it is more likely our component uses ARMv4.

Table 1: Filter and mask for determining Thumb instructions. Together with the PUSH instruction, they are used for recovering more code from the target firmware.

Instruction	Filter (bits) 7 6 5 4 3 2 1 0	Mask (bits) 7 6 5 4 3 2 1 0
SUB	1 1 1 1 1 0 1 0	0 0 0 1 1 0 1 0
LDR	1 1 1 1 1 0 0 0	0 1 0 0 1 0 0 0
MOVS	1 1 1 1 1 0 0 0	0 0 0 1 1 0 0 0
BL (offset low)	1 1 1 1 1 0 0 0	1 1 1 1 1 0 0 0
BL (offset high)	1 1 1 1 1 0 0 0	1 1 1 1 0 0 0 0

loaded. Code is created only at the addresses which were previously logged.

Next, *instructions patterns* which are used in the *prologue of functions* are identified, and used to recover more code. Most of the firmware code uses the Thumb instruction set, therefore we target the following instructions:

- PUSH [...] SUB
- PUSH [...] BL
- PUSH [...] LDRr
- PUSH [...] MOVS

The instructions are represented on 2 bytes (except BL) and are identified based on the opcodes they use. The PUSH instruction has the opcode 0xb5 (if the Link Register (LR) is also pushed) or 0xb4 (without LR), followed by a byte representing which of the registers R0-R7 are used. For the other instructions, masks and filters are defined, in order to determine if the instruction following a PUSH is indeed one of the previously mentioned ones. Table 1 shows the values defined. The filter determines which bits are of interest, and the mask determines what the value of those bits should be. The branch with link instruction (BL) is represented on 4 bytes, first containing the upper 11 bits of the target address, then the 11 bits of the lower half of the address.

The result is a fairly clean version of the firmware with code made at relevant addresses. However, due to misinterpretation between ARM and Thumb, some parts of the code were created as ARM, when they were in fact Thumb, and thus are not recognised as valid functions, and this behaviour is not rectified by setting the Thumb segment register.

In order to deal with this, we wrote an IDAPython script, which determines the changes in the Thumb segment register, undefines the sections which are ARM code, and re-sets the Thumb register. It then iterates through the whole ROM segment and logs the addresses where the changes from Thumb to ARM still happen, and these can then be used to manually inspect the code around that address. Some parts of the recovered code will belong to genuine functions, but the end of the functions is set at an earlier address. This is solved by adding the chunks of instructions to the original functions. The code navigation bar helped in identifying end areas for the code being inspected. We found that the script greatly aided the process by targeting specific areas. Using these methods,

```
e_lwz      rA, DMR
se_mtctr   rA
se_bctrl
```

where rA represents a general register, and DMR represents a Direct Memory Reference.

Figure 3: Load address to control register pattern and branch to address in control register.

60.21% of the firmware was converted into explored addresses[4]. 17.52% of the firmware is a block of 0xFFs, therefore only 22.27% of the firmware code could not be automatically disassembled and requires further manual inspection.

Disassembling PowerPC. For the PPC firmware, we have to make sure the IDA settings are correct. We set the processor as big-endian PPC, select the Signal Processing Engine (SPE) instruction set and enable Variable Length Encoding (VLE). IDA has support for a few PPC devices, and we select mpc5xx, as this is the generic version of the board our ECU uses.

For this ECU we were able to obtain the flash and the RAM contents. Therefore, we have to create the RAM segment, and also the IO segment. From the datasheet we know that the reset vector is located at address 0x04, and this is the point from where we start our analysis. The vector points to ROM address 0x160. Once we create code at that address, we analyse the functions that spawn from there (in our case, 94 new functions were created). We notice that 66% of the functions start with the se_mflr r0 instruction, which moves the contents of the Link Register into register 0. We wrote a script which looks for the instruction opcode and operand, 0x00 0x80, creating 5063 new functions with this method. As with the ARM code, some of the functions have the ending set before the function actually ends, leaving some instructions on the outside. Therefore we need to manually adjust these. As functions tend to end with a se_blr instruction, we can easily determine if instructions which do not belong to any function should, in fact, be part of the function that exists before them. Otherwise, we mark the set of instructions as their own function, and inspect the code around it. Next, we observe that indirect jumps are used (a control register is used in controlling the flow of the firmware). A set of three instructions is used (Figure 3); first, a general register is loaded with an address from the ROM. Then, the control register is loaded with the address the register points to. The flow of the program then branches to the address contained in the control register. Therefore, by extracting the addresses referenced by the Direct Memory Reference (DMR), we can discover more functions of the firmware.

Using this method 52.77% of the ROM was recognised as explored addresses. The firmware contains two large blocks of bytes with the value 0xFF, towards the end of the ROM address space. The blocks represent 23.21% of the size of the firmware. Therefore, 24.02% of the firmware needs to be further manually inspected.

Disassembling Infineon TriCore. The Infineon TriCore ECU was the most challenging out of the components we worked with. The documentation for the MCU was sparse, and the Infineon website did not contain the user manual for the TC1793 MCU, and instead linked to the user manual of the TC1798 MCU. The two MCUs are from the same family, though comparing the datasheets of the

[4]IDA marks an address as explored if it is correctly recognised as code or data.

two revealed a few minor differences, such as the TC1793 having fewer Analog-to-Digital Converters and fewer General Purpose Input/Output lines. Conflicting information was found in datasheets and user manuals with respect to the location of the reset vector, with some specifying the reset vector is at offset `0x0` in the flash memory, and others placing the Interrupt Table Vector at that address. Regardless, our firmware had a bank of zeroes as the first 16393 bytes, so neither sources seemed to be correct.

Furthermore, while IDA does have support for TriCore, none of the device profiles match the TC1793/TC1798. Therefore, we start off with a device IDA does know, TC1797, remove all segments which do not match the specification of the TC1793, and create the correct ones. The flash memory of the TC1793 is divided between two address ranges, each of 2 Kbytes, and so we need to split and load the firmware at the appropriate offsets. Once the memory map was correctly recreated and the firmware loaded, we can start considering retrieving functions.

Unlike ARM and PPC, TriCore has eliminated the need for function prologues and epilogues, through code optimisation [Technologies 2003]. Therefore, searching for opcodes would not help, for this particular architecture. We used the forceful code creation method, combined with manual inspection of code. While more time consuming, the method has successfully led to 54% of the total address space to be recognised as explored. The firmware contained three large blocks of byte values `0xC3`, amounting to 11.81% of its size. 34% of the firmware required further inspection.

3.3 Control Flow Graph Extraction

This section describes how we extract the CFG, in order to later use it with the fuzzer. In order to create an accurate CFG, we use an intermediary object, named a Control Flow List (CFL). The CFL is obtained by calling the `Flowchart()` function provided by the IDAPython Application Program Interface (API). The function returns a list of `BasicBlock` objects, which contain information about the start and end address of the basic block, as well as two lists, one being the predecessors of the basic block, and one containing the successors.

Once satisfied the CFL is accurate, it can be used to create the graph, using the Python **`igraph`** module. The start and end address of the basic block are added as vertex attributes. The vertices will later be labelled with a static data value, if the branch of the basic block is dependent on a comparison with it (more details in Section 4.1). The edges will be labelled with the condition that needs to be met in order to go down that path. Edges that point from a vertex to itself are ignored.

CFG Extraction from ARM Firmware. As previously mentioned in Section 3.2, calls to Thumb addresses are not correctly handled by IDA. It does not correctly identify a call to a Thumb function as the end of a basic block. Therefore, the CFG needs to be amended, such that it more accurately represents the firmware. Thumb functions are called using a pattern of three instructions, as represented in Figure 4. The value of the DMR determines the address of the Thumb function.

Given a CFL created strictly with the results returned by the `Flowchart()` function, and the previously identified Thumb function calls, we amend the CFL to split the basic blocks containing

```
LDR rA, DMR
MOV LR, PC
BX rA
```

where `rA` represents a general register, and DMR represents a Direct Memory Reference.

Figure 4: Pattern for branch to THUMB function, from ARM code.

calls to Thumb functions, adding the Thumb function to the successors of the first new basic block, and the second new basic block as a successor to the first one.

CFG Extraction from PowerPC Firmware. As discussed in Section 3.2, some functions are accessed by being referenced through a DMR. The `Flowchart()` function of the IDAPython API does not recognise these as part of the control flow of the firmware.

Similarly to the procedure used for the ARM firmware, we create the CFL by calling the `Flowchart()` function, then we look for the pattern of instructions presented in Figure 3. We extract the address referenced by the DMR, and amend the basic blocks to include the function correctly.

CFG Extraction from Infineon TriCore Firmware. The CFG extraction is straightforward for the TriCore firmware, as there are no functions loaded through DMR accesses. The `Flowchart()` function returns the correct CFG, and we only need to create the `Graph` object, based on its information. Figure 5 shows the extracted CFG of *one* function, from the Engine Control Module ECU (Infineon TriCore). It shows the labelling of the vertices with basic block information, such as start and end addresses and last instruction of the basic block, as well as static data used in deciding the control flow of the firmware.

4 FUZZING ELECTRONIC CONTROL UNITS

The role of the work described in Section 3, is to enable the design of a CAN fuzzer which uses *static data* extracted from the ECU firmware to create CAN messages. This section describes how the data is extracted and the design of the fuzzer.

Tools and Setup. The diagram in Figure 6 shows the hardware setup required by the fuzzer.

In order to communicate with an ECU we used the PeakCAN USB interface[5] (2) together with the OBD-II connector[6] (3), which are connected to the laptop (1) running the fuzzer. We used a breadboard (4) for wire management. The PeakCAN is connected to the target ECU though the CAN pins on the OBD-II connector (3) and on the device under test (6). An external power supply (5) provides the required 12V. The PCAN Python API provides the module **`PCANBasic`**, which handles the initialisation and configuration of communication channels, as well as transmitting and receiving CAN messages. We continue using the **`igraph`** module for manupulating and working with the program CFG and we use the **`pickle`**[7] module for storing the graph to disk.

[5]PCAN-USB: CAN Interface for USB (`https://www.peak-system.com/PCAN-USB.199.0.html?&L=1`)
[6]PCAN-Cable OBD-2: CAN-OBD-2 Diagnostics Cable (`https://www.peak-system.com/PCAN-Cable-OBD-2.273.0.html?&L=1`)
[7]Python module for serialising and de-serialising object structures (Python `pickle` documentation: `https://docs.python.org/3/library/pickle.html`).

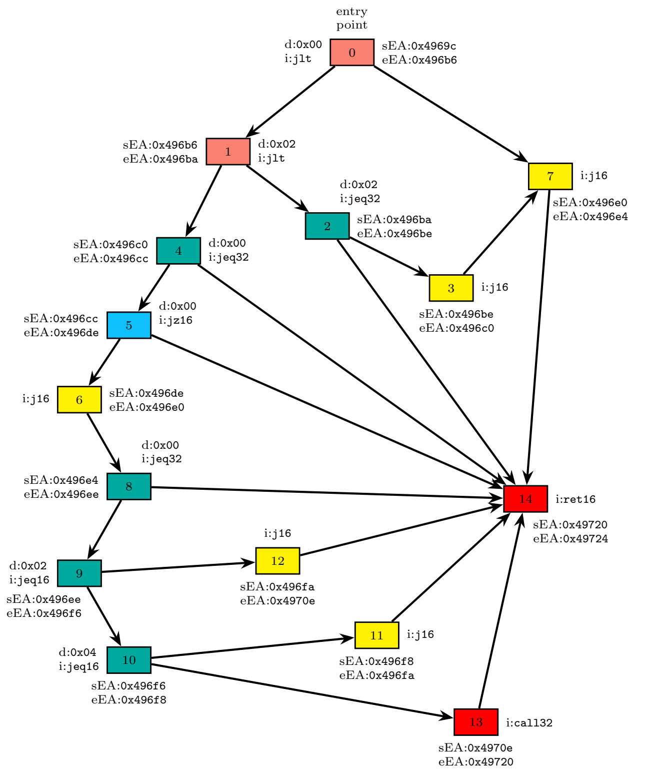

Figure 5: Extracted CFG, representing the basic blocks of *one* function. Each basic block was tagged with information about the start address (sEA), end address (eEA), static data used in comparisons (d) and the last instruction of the basic block (i). The function is from the ECM ECU, Infineon TriCore architecture

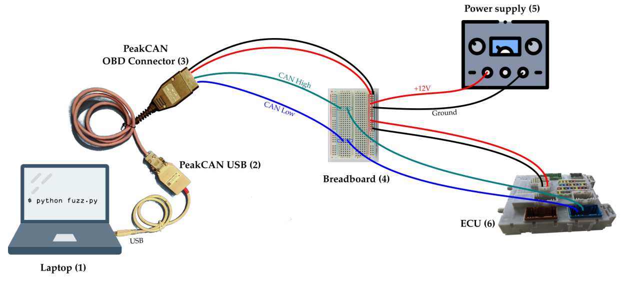

Figure 6: Fuzzer hardware setup.

4.1 Data Extraction

The first step towards creating the fuzzer is the data extraction step. This process still requires the disassembly, as we will use it to label the firmware CFG with additional information, but after this step, any further work can be done without the need of IDA.

Control Flow Graph Tagging. The CFG is labelled with the static data values used in comparisons by looking for instruction patterns.

Table 2: Example of instruction patterns for identifying comparisons with static data for the three architectures studied. `IMMED` refers to an immediate value, `rA` and `rB` are general registers and `ADDR` is a ROM address within the firmware.

ARM Architecture		

`CMP rA, IMMED`
`BEQ ADDR`
	`MOVS rA, IMMED`
	`CMP rB, rA`
	`BGT ADDR`	

PPC Architecture		

`e_lis rB, IMMED1`
`e_add16i rB, rB, IMMED2`
`se_cmpl rA, rB`
`se_bne ADDR`
	`e_cmpi cr0, rA, IMMED`
	`se_bne ADDR`
		`se_cmpli rA, IMMED`
		`se_bge ADDR`

Infineon TriCore Architecture		

`jne32 rA, IMMED, ADDR`	`mov16 rA, IMMED`
	`jge.u rA, rB, ADDR`
		`jz16 rA, ADDR`

These patterns are architecture-dependent, as shown in Table 2. The data is then set as an attribute for the vertex corresponding to the basic block. The edges of the graph are labelled with the instruction the program control would take in order to go down that path. For this purpose we define an `Antonyms` dictionary, through which we map the opposite instruction for each branch or jump instruction we encounter (e.g. BEQ – BNE). The instructions will later be used by the fuzzer in determining what values could a byte take, while still respecting the condition of the branch or jump instruction.

Lastly, the vertices are also labelled with *probability* and *weight* metrics, and the edges with the *probability* metric. Inspired by VUzzer [Rawat et al. 2017], we calculate the probability that a basic block will be reached. For this, we take as starting point a vertex which has an in-degree of 0 and we identify all vertices that can be reached from it, by performing a breath first search. Then we iterate over the search result and calculate the probabilities accordingly. The weight is the inverse of the probability, for vertices. The probability of the edges is dependent on the out-degree of the originating node:

$$p_{e(i,j)} = \frac{1}{out\_deg(i)} \tag{1}$$

where $p_{e(i,j)}$ is the probability of the edge between vertices i and j, and $out\_deg(i)$ is the out-degree of vertex i. The module `sympy` is used to solve the probabilities of vertices which are part of a cycle. The metrics are used by the fuzzer to determine which byte of the payload to fuzz. Bytes corresponding to basic blocks with high weights will be given priority.

Figure 7 shows the CFG for *one* function, from the BCM firmware, having its vertices tagged with basic block start address, static data that influences the branch condition, probability and weight, and its edges tagged with the probability and the instruction satisfying the specific paths.

Forming Input Seeds for the Fuzzer. In creating the data chains we take advantage of the small payload of the CAN frames, by

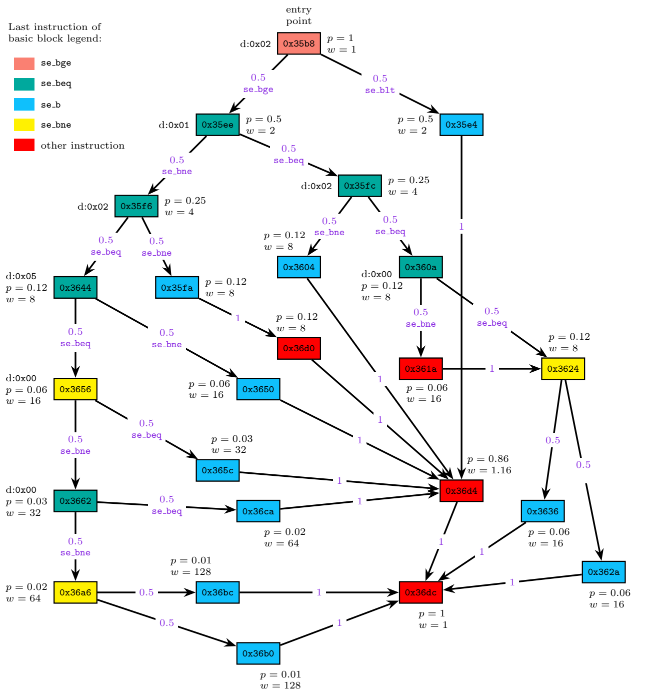

Figure 7: Example CFG of *one* function (from the BCM, PPC architecture), after its basic blocks have been tagged with probabilities (p) and weights (w), by applying (1) to each basic block, with the static data on which the control flow is decided (d) and with the probabilities for the paths (values on the edges). The vertex label is the start address of the basic block.

creating payloads with at most 8 bytes. We assume that the payload bytes are processed sequentially, in consecutive basic blocks. Algorithm 1 describes the process of extracting the data chains. We iterate through the vertices of the CFG and look for those which have been tagged with data (function `startEA_with_data`). These will be the starting points for the exploration algorithm. For each starting point, we then perform a depth-first search of the CFG, through the function `find_paths` (line 6). We begin at a given vertex `start_vertex`, up to a maximum depth of `max_depth`, and search for as long as comparisons with static data occur in consecutive vertices. The function stores a list of vertex sequences that have been explored in `paths_found`. This leaves us with a list of vertex sequences stored in `vids`. In lines 9–13, we extract the static data associated with those vertices and build the list `data`. Each path in `vids` has a corresponding path in `data`.

After the chains have been extracted, a list of `Payload` (Listing 1) objects is created. Each instance has as attributes the data chain list, the vertices ids list corresponding to the data, a list which will keep track of which bytes have been modified, as well as a probabilities list and a weights list for the basic blocks corresponding to the data.

Algorithm 1 `explore_cfg`

```
Input: $cfg$                                   # CFG of firmware analysed
Output: $vids, data$ # $vids$ contains the lists of vertex indexes, $data$ contains the lists of data
    payloads
 1: $data\_startEAs \leftarrow$ startEA_with_data($cfg$)   # retrieve a list of vertices where
    comparisons with static data occurs
 2: $vids \leftarrow$ list()                               # declare new list
 3: for $start\_vertex$ in $data\_startEAs$ do
 4:     $explored\_vids \leftarrow$ list()                 # declare new list
 5:     $paths\_found \leftarrow$ list()                   # declare new list
 6:     find_paths($cfg, start\_vertex, explored\_vids,$
        $paths\_found, max\_depth$)
 7:     $vids$.extend($paths\_found$)                      # add all found paths to list
 8: end for
 9: $data \leftarrow$ list()                               # declare new list
    # for each path of vertices found, create a corresponding list of the static data values: #
10: for $path$ in $vids$ do
11:     $path\_data \leftarrow$ list()                     # declare new list
12:     for $vertex$ in $path$ do
13:         $path\_data$.append($cfg.vs[vertex]$["data"])  # extract data from CFG
14:     end for
15:     $data$.append($path\_data$)          # add the list with the data sequence to the list
16: end for
17: return ($vids, data$)
```

Listing 1: Definition of the Payload class and its member attributes

```
class Payload(object):
    def __init__(self, payload, vids, probability_score,
         weight):
        self.payload = list(payload)
        self.vids = list(vids)
        self.fuzzed_bytes = [ False for _ in len(self.
            payload) ]
        self.probability_score = list(probability_score)
        self.weight = list(weight)
```

4.2 Fuzzer Design

The following section is concerned with the design and implementation of the fuzzer. It presents the heuristics for the seed transformation, as well as the options the program has implemented.

Fuzzer Prerequisites. The program takes as input the following arguments:

```
> python2 effCAN.py <arch> <suffix> <cfg-file> <queue/
    payload> <USB-device> [resume]
```

where:
`<arch>` is the architecture of the target ECU firmware: `arm/ppc/tricore` (mandatory);
`<suffix>` is the suffix of the vertices and data file; it helps distinguish files if multiple ECUs with the same architecture have been analysed – (mandatory);
`<cfg-file>` is the CFG file of the firmware, saved in pickle format – (mandatory);
`<queue/payload>` is the method of fuzzing; the program can either fuzz one byte for each payload in the queue and wrap around or can fuzz all bytes in each payload, then move on to the next payload (mandatory);
`<USB-device>` is the number of PCAN interface, as seen on the host computer (mandatory);
`[resume]` signals to the fuzzer it should continue from the last known state (optional).

Based on the arguments specified, the program looks for the following files:
`payloads_<arch>_<suffix>.pickle`: the file contains the list of `Payload` objects;
`payloads_resume_<arch>_<suffix>.pickle`: the file contains the list of fuzzed `Payload` objects; used in order to resume fuzzing from a saved state; this file is sought only if the `resume` option is enabled;
`ids_<arch>_<suffix>.pickle`: the file contains the CAN IDs the fuzzer will send messages on; if the file does not exist, it will try all possible IDs.

Input Transformation. For each branch/jump instruction, a function is defined that will choose a random value such that the condition is still respected. This is used in the input transformation.

The fuzzing process is iterative and works as follows. Given a queue of `Payload` objects, the first value is popped, and the payload is subject to transformation. If the program is run with the *queue* option, one of the bytes of the payload is chosen to be modified. A new value is chosen, such that the instruction condition is respected. The new message is sent over the CAN interface, and a new `Payload` object, with the modified payload, is added to the end of the queue. If the *payload* option is chosen, all the bytes of the payload are subject to transformation, and this whole new value is sent over CAN to the ECU, then added to the queue.

The transformation function operates on one `Payload` object at a time. It looks if there are any bytes which have not yet been fuzzed, then looks at the probabilities of these bytes. It will choose the byte with the lowest probability, and it will choose a random value that still respects the condition of the instruction that defined it. For example, if the instructions sequence was

```
CMP rA, 0x40
BGT ADDR
```

and we are on a path that takes the branch, the value has to be in the interval $(0x40, 0xFF]$. If a new value cannot be chosen, the initial value of the payload is retrieved and the process is repeated[8]. Once all the bytes have been fuzzed, in the *queue* mode, the `fuzzed_bytes` list is reset. For the *payload* mode, the `fuzzed_bytes` list is not used, as all bytes are fuzzed in each iteration.

Crash Detection. As discussed in Section 1, knowing what a *crash* means with respect to an ECU is difficult, due to the limited output and feedback they provide. Furthermore, as ECU firmware logic is mostly driven by interrupts, reproducibility of a result is problematic. This is due to the difficulty of guaranteeing the ECU is in exactly the same state on two different runs.

Our fuzzer uses two methods of detecting crashes, depending on the information available about the ECU: message *timeout* or the *UDS Tester Present* service. Both methods detect crashes based on the CAN traffic the ECU under testing outputs. Before the fuzzing process starts, the program listens for all the messages the ECU sends data on and keeps a record of them. While sending fuzzed payloads, it also listens to incoming traffic. If a message on a new CAN ID is recorded, it will flag this up for inspection.

[8]There may be cases in which a byte cannot get another new value, e.g. if the immediate value of a `BLT` instruction has been fuzzed to `0x0` in a previous iteration, the values interval would be $[0, 0)$. In order to avoid this, we retrieve the original value of the payload and operation can resume normally.

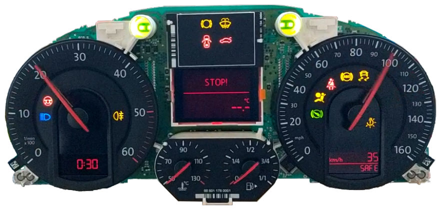
Figure 8: Volkswagen IPC in testing.

For the timeout method, if the ECU stops sending messages for a specified time (parameter), it considers the ECU has crashed. For the Tester Present service, the CAN ID for UDS needs to be known. After each fuzzed payload is sent to the ECU, a request is made to the Tester Present service (`0x3E`). If the service responds, regardless of whether it has a positive or negative reply, it means the ECU is still functioning. If there is no response, within a specified time, it considers the ECU has crashed. The delay in response is also logged, and this can be analysed and potentially provide clues as whether certain messages introduce a greater delay. This could mean the ECU was in a possibly unexpected state.

5 EVALUATION AND RESULTS

The evaluation process consisted of two steps: first, we tested whether the input seeds we extracted from the firmware are relevant to the appropriate ECUs, without any transformation being applied; we then ran the fuzzer on the ECUs mentioned in Section 3.

For the first step, after we extracted the byte sequences, we tested the validity of the data chains. As previously mentioned, working with ECUs is tricky, as CAN communication is unidirectional and there is no feeback for a sent message. The best target for testing was the Volkswagen IPC, as it has a multitude of visual outputs, in the form of gauges and indicators. As we did not know the CAN IDs the ECU was programmed to listen to, we sent the data chains extracted from the firmware on all possible IDs, without any further modification. We visually monitored the IPC and noted if any of the gauge needles moved, if indicators changed status or if the information of the display was modified. The test was successful, as seen in Figure 8. The sent CAN messages did indeed trigger the various lights the panel had and moved the speed and the tachometer needles in a rapid succession. Therefore, our initial hypothesis, that *data on which the control flow of the firmware is decided is more likely to be meaningful to the ECU than randomly chosen messages*, is validated.

We then tested the fuzzer on the three ECUs, and EffCAN produced crashes in two of them. We reiterate that due to the limited feedback the ECUs provide, the results cannot always be explained, and we can only hypothesise about them. During the tests we have observed unexpected behaviours from the ECUs, where they stopped communicating via the CAN bus. This type of behaviour can be very dangerous, especially in the case of safety-critical ECUs, such as the ECM. For most situations, rebooting the ECU enabled it to resume normal function. However, for the BCM, on three separate occasions, the ECU did not work as expected after reboot. It required a cool-off time of about 2-4 hours, but we cannot explain the need for it. The ECU Printed Circuit Board does not have any large capacitors, the one possible explanation we considered feasible for the ECU to be able to retain its state for such a long time. The results were not reproducible by replaying the same set of CAN messages to the BCM. This is not highly surprising; as mentioned previously, the ECU is driven by interrupts, therefore making sure it is in the exact same state twice is difficult without execution traces.

Nonetheless, our results are highly promising. We demonstrate that data extracted from the ECU firmware is more meaningful than randomly chosen data, and that fuzzing with this data as input seeds leads to crashes and unexpected behaviours. This research lays the ground work for what we hope will be further research into the security of the ECUs, and creating automated frameworks and tools for analysing the firmware of electronic components is a crucial part of assessing the security of ECUs.

Future research directions. While the results are positive, there are areas that can be further developed. Firmware analysis still has a number of open research questions, such as control flow recovery, function identification and data structure recovery, the former two upon which we touch in this article. Handing indirect function calls and indirect branch instructions are the main challenge for this, especially in cases where control flow is transferred via function pointers or the destination address may be dynamically computed [Di Federico et al. 2017]. Function pointers complicate function detection as well, therefore a heuristic cannot only rely on explicit function calls. Improvements in these areas will allow for a more accurate CFG recovery, having a better representation of the firmware analysed, and therefore improving any heuristics inferred from said analysis.

Evidently, executing the firmware we are trying to test is a central part of a fuzzer. In our solution, we chose a hardware-in-the-loop system, but this is not a highly scalable option, as it requires additional hardware to parallelise the procedure. Given the issues with full emulation discussed in the *Challenges* section, significant engineering commitments would be required to bring ECU support to existing emulators. Partial emulation could cover some of the pitfalls of full emulation. For example, Frankenstein [Ruge et al. 2020] solves the issue of unknown memory map of embedded devices by emulating the firmware and delegating any memory map functionality back to the device, such that it is retrieved straight from an execution run. Nevertheless, their main limitation of emulating the firmware is the support for a limited number of architectures.

Finally, one of the most significant improvements that could be brought is identifying the cause of the crashes. For our solution, we used physically observable behaviours or gaps in the communication with the device. However, the feedback loop can be improved by identify the root cause of the events and, as we mentioned earlier, by establishing what the state of the device is when a crash occurs, which would also improve reproducibility. Using instrumentation in order to track the specific execution path taken when provided with an input would also increase the performance of the fuzzer, as this would allow for better control flow coverage, and enable a more efficient exploration of the firmware functionality. However, this solution needs to deal with the pollution caused by additional paths that time-based interrupts would introduce.

6 CONCLUSION

Most state-of-the-art fuzzers from the literature have an emulation component, or some form of feedback from executing the program with a given input, which informs the next iteration of input transformation. Our fuzzer purely relies on static analysis, as obtaining hardware execution traces from ECUs is a complex, if not impossible, task. Furthermore, good emulators for automotive MCUs are not available, and the effort involved in developing them is high. The tools for automotive-specific research are not yet mature enough to allow for fully automated, large scale analyses, which is why we believe our approach is a good step in beginning to fill this large gap.

In this paper, we have proposed a method for analysing automotive ECU firmware, by extracting the control flow of the firmware into a CFG structure. This allows tools to be built, regardless of the underlying architecture of the ECU. Due to the large number of architectures ECUs are built on, the sparsity of documentation and the limited number of documented attempts of ECU firmware analysis, this has been a challenging task, and we believe it presents a high entry barrier for possible researchers having an interest in the field. As there is only one other description of analysing ECU firmware, we strongly believe in the added value of the explanations we have provided.

Furthermore, we have presented a fuzzer for ECUs, which communicates with a target component via the CAN bus. The fuzzer uses data extracted from the firmware running on the ECU and is guided by the CFG of the firmware in its seed transformation process. To our knowledge, this is the first automotive fuzzer available in the literature that relies on something more complex than randomly choosing the bytes of the messages to send. Our tests show that indeed this methodology yields better results than the purely random strategy.

REFERENCES

[n.d.]. Tools for Automotive Repairing. http://www.usprog.ru/index.php/en/news/usp.html

2000. CDC 32xxG – Car Dashboard Controllers. http://pdf.datasheetcatalog.com/datasheet/MicronasIntermetall/mXvsrxz.pdf

Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 177–189.

Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *23rd USENIX Security Symposium (USENIX Security 14)*. 845–860.

Stephanie Bayer and Alexander Ptok. 2015. Don't Fuss about Fuzzing: Fuzzing Controllers in Vehicular Networks. (2015).

David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. 2011. BAP: A binary analysis platform. In *International Conference on Computer Aided Verification*. Springer, 463–469.

Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. 2006. Detecting self-mutating malware using control-flow graph matching. In *International conference on detection of intrusions and malware, and vulnerability assessment*. Springer, 129–143.

Jan Van den Herrewegen and Flavio D. Garcia. 2018. Beneath the Bonnet: A Breakdown of Diagnostic Security. In *23rd European Symposium on Research in Computer Security (ESORICS 2018), Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11098)*. Springer, 305–324. https://doi.org/10.1007/978-3-319-99073-6

Valgrind Developers. 2017. Valgrind supported architectures. http://www.valgrind.org/info/platforms.html

Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. rev. ng: a unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*. 131–141.

Daniel S Fowler, Jeremy Bryans, and Siraj Shaikh. 2017. Automating fuzz test generation to improve the security of the Controller Area Network. (2017).

Daniel S Fowler, Jeremy Bryans, Siraj Ahmed Shaikh, and Paul Wooderson. 2018. Fuzz Testing for Automotive Cyber-Security. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 239–246.

Flavio D. Garcia, David Oswald, Timo Kasper, and Pierre Pavlidès. 2016. Lock It and Still Lose It - On the (In)Security of Automotive Remote Keyless Entry Systems. In *25nd USENIX Security Symposium (USENIX Security 2016), to appear*. USENIX Association.

Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, et al. 2019. Toward the Analysis of Embedded Firmware through Automated Re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*. 135–150.

Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. 2012. Juxtapp: A scalable system for detecting code reuse among android applications. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 62–81.

Christopher Hicks, Flavio D Garcia, and David Oswald. 2018. Dismantling the AUT64 Automotive Cipher. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2018), 46–69.

ISO. 2013. *14229: 2013 – Road Vehicles – Unified diagnostic services (UDS)*. Standard. International Organization for Standardization.

Johannes Kinder and Helmut Veith. 2008. Jakstab: A static analysis platform for binaries. In *International Conference on Computer Aided Verification*. Springer, 423–427.

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

Charlie Miller and Chris Valasek. 2015. Remote Exploitation of an Unaltered Passenger Vehicle. http://illmatics.com/Remote%20Car%20Hacking.pdf

Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. 2018. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices.. In *NDSS*.

Minh Hai Nguyen, Thien Binh Nguyen, Thanh Tho Quan, and Mizuhito Ogawa. 2013. A hybrid approach for control flow graph construction from binary code. In *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, Vol. 2. IEEE, 159–164.

Pranav Patki, Ajey Gotkhindikar, and Sunil Mane. 2018. Intelligent Fuzz Testing Framework for Finding Hidden Vulnerabilities in Automotive Environment. In *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*. IEEE, 1–4.

Nam H Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2010. Detection of recurring software vulnerabilities. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 447–456.

Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing.. In *NDSS*, Vol. 17. 1–14.

Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. 2020. Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets. In *29th USENIX Security Symposium (USENIX Security 20)*. 19–36.

Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing functions in binaries with neural networks. In *24th USENIX Security Symposium (USENIX Security 15)*. 611–626.

Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *NDSS*, Vol. 16. 1–16.

Infineon Technologies. 2003. TriCore Compiler Writer's Guide. https://www.infineon.com/dgdl/inf0010_v1_4Dec2003_1.pdf?fileId=db3a304412b407950112b40f8aad1423

Roel Verdult, Flavio D. Garcia, and Barış Ege. 2015. Dismantling Megamos Crypto: Wirelessly Lockpicking a Vehicle Immobilizer. In *22nd USENIX Security Symposium (USENIX Security 2013)*. USENIX Association, 703–718.

Pei Xia, Makoto Matsushita, Norihiro Yoshida, and Katsuro Inoue. 2014. Studying reuse of out-dated third-party code in open source projects. *Information and Media Technologies* 9, 2 (2014), 155–161.

Michal Zalewski. 2014. American fuzzy lop. http://lcamtuf.coredump.cx/afl