# AUTOMOTIVE FIRMWARE EXTRACTION AND ANALYSIS TECHNIQUES

by

Jan Van den Herrewegen

A thesis submitted to
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
College of Engineering and Physical Sciences
The University of Birmingham
March 2021

# Abstract

An intricate network of embedded devices, called Electronic Control Units (ECUs), is responsible for the functionality of a modern vehicle. Every module processes a myriad of information and forwards it on to other nodes on the network, typically an automotive bus such as the Controller Area Network (CAN). Analysing embedded device software, and automotive in particular, brings many challenges.

The analyst must, especially in the notoriously secretive automotive industry, first lift the ECU firmware from the hardware, which typically prevents unauthorised access. In this thesis, we address this problem in two ways: (*i*) We detail and bypass the access control mechanism used in diagnostic protocols in ECU firmware. Using existing diagnostic functionality, we present a generic technique to download code to RAM and execute it, without requiring physical access to the ECU. We propose a generic firmware readout framework on top of this, which only requires access to the CAN bus. (*ii*) We analyse various embedded bootloaders and combine dynamic analysis with low-level hardware fault attacks, resulting in several fault-injection attacks which bypass on-chip readout protection.

We then apply these firmware extraction techniques to acquire immobiliser firmware by two different manufacturers, from which we reverse engineer the DST80 cipher and present it in full detail here. Furthermore, we point out flaws in the key generation procedure, also recovered from the ECU firmware, leading to a full key recovery based on publicly readable transponder pages.

# ACKNOWLEDGEMENTS

As Margaret Hungerford expressed eloquently, beauty lies in the eyes of the beholder. So, first and foremost, I would like to thank you, the reader, for taking the time out of your busy life to pick up this thesis and, however briefly or thoroughly, read (part of) it.

Along the way I have had the privilege to meet and work with many exceptional people. First of all, I would like to thank my supervisor turned friend, Flavio Garcia, without whom I would have never started a PhD, let alone finish it. Thank you for the many interesting discussions, coffee breaks and most of all the occasional life advice. Equally, my gratitude goes out to my co-supervisor David Oswald, who always had an answer to my (at times naive) questions and never failed to provide me with a new insight to tackle problems.

I extend my gratitude to my examiners, Tom Chothia and Colin O'Flynn, who kindly gave up their valuable time to read and assess this thesis.

To my office mates and friends throughout the department, Andreea, Kit, Chris, Sam and Richard - thank you for the (at times too) pleasant work environment. To my new family and friends in what they would refer to as 'gods country' Franz, Emilio, Charlie, Fred, the Jacks and Sams, and to my friends back home, Luigi, Seba, Martino, Pedro, Florrie - thank you for keeping me sane and bringing some perspective to the long and at times arduous PhD journey. To my coaches, Rob, Jon and Burnsy, thank you for all your advice and patience.

Finally, I would like to thank my family for their unconditional support. Inge and Tim for putting up with their little brother and teaching me the ways of life. And my parents, Lut and Piet, for being so supportive of any decision I take, be it in sport, studies or life.

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

## List of Acronyms

| | |
|---|---|
| **BCM** | Body Control Module |
| **BL** | Bootloader Enable |
| **CAN** | Controller Area Network |
| **CBS** | Critical Bootloader Section |
| **CCP** | Can Calibration Protocol |
| **CFG** | Control-Flow Graph |
| **CRP** | Code Readout Protection |
| **DoS** | Denial-of-Service |
| **DST** | Digital Signature Transponder |
| **ECU** | Electronic Control Unit |
| **EEPROM** | Electrically Erasable Programmable Read-Only Memory |
| **EMFI** | Electromagnetic Fault Injection |
| **GIAnT** | Generic Implementation ANalysis Toolkit |
| **HF** | High Frequency |

| | |
|---|---|
| **IC** | Integrated Circuit |
| **IDS** | Intrusion Detection System |
| **ISA** | Instruction Set Architecutre |
| **ISO** | International Organization for Standardization |
| **ISO-TP** | ISO Transport Layer |
| **JTAG** | Joint Test Action Group |
| **KWP** | Keyword Protocol 2000 |
| **LF** | Low Frequency |
| **LFSR** | Linear Feedback Shift Register |
| **LTE** | Long Term Evolution |
| **MAC** | Message Authentication Code |
| **MMU** | Memory Management Unit |
| μC | Microcontroller |
| **OBD-II** | On-Board Diagnostics |
| **OEM** | Original Equipment Manufacturer |
| **OS** | Operating System |
| **PCB** | Printed Circuit Board |
| **PKE** | Passive Keyless Entry |
| **PKES** | Passive Keyless Entry and Start |
| **POR** | Power-On Reset |
| **RAM** | Random Access Memory |
| **RFID** | Radio Frequency IDentification |
| **RKE** | Remote Keyless Entry |
| **RNG** | Random Number Generator |
| **ROP** | Return-Oriented Programming |
| **RTOS** | Real-Time Operating System |

| | |
|---|---|
| **SPI** | Serial Peripheral Interface |
| **SoC** | System-on-Chip |
| **SUT** | System Under Test |
| **SWD** | Serial-Wire Debug |
| **SWIM** | Single Wire Interface Module |
| **TCU** | Telematics Control Unit |
| **TPMS** | Tire Pressure Monitoring System |
| **TI** | Texas Instruments |
| **TP 2.0** | Transport Protocol 2.0 |
| **UART** | Universal Asynchronous Receiver Transmitter |
| **UDS** | Unified Diagnostic Services |
| **V-FI** | Voltage-Fault Injection |
| **VAG** | Volkswagen Auto Group |
| **VIN** | Vehicle Identification Number |
| **VLE** | variable length encoded |
| **XCP** | Universal Measurement and Calibration Protocol |
| **XOM** | eXecute-Only-Memory |
| **OCD** | On-Chip Debug |

# CHAPTER 1

# INTRODUCTION

Vehicles are ubiquitous in modern day life. In 2019, the automotive industry produced a total of 92.8 million vehicles worldwide [59]. Across Great Britain, on average 68% of commuters did so by car in 2019, while 79% of freight was transported by road in 2018, pointing at the widespread use of both passenger vehicles and trucks [52]. With most manufacturers to a greater (e.g., self-driving cars) or lesser (e.g., lane-assist technology) extent moving towards more autonomous vehicles, inevitably the technological aspect of a vehicle will only increase. However, unlike other more transient electronic appliances, such as phones or personal computers, cars are designed to stay in use for decades. Hence, manufacturers must strike a balance between incorporating the newest technologies, while still safeguarding the vehicle's longevity. This also manifests itself in a systems security point of view: the average automobile lasts over a decade [58], which is considered a long time in the fast-moving field of computer security. Over the years, the automotive industry has continuously introduced new safety procedures in order to make their vehicles safer (e.g., the three-point seat belt [218]). However, with a modern vehicle nowadays resembling an electronic appliance more than a purely mechanical one, the industry must now equally focus on securing its fleet from a new type of adversary, the connected attacker.

A host of interconnected ECUs determines a vehicle's capabilities, each of these embedded automotive computers with their own interfaces and peculiarities. Millions of lines of code [35] interact seamlessly and hide the vast complexity of a modern automobile. What used to be a closed, purely mechanical system is now a metropolis of interacting embedded devices, inevitably raising security concerns. With connected components (e.g., Bluetooth, 4G, WiFi) now an integral part of the automotive ecosystem, the internal vehicular network can no longer be treated as a closed system. This brings along security implications not only for ECUs connected to the outside world, but for all other ECUs inside the car. After all, if an external attacker compromises any remote component they can create a bridge to the internal vehicular network (see e.g., [36, 126]). On top of that, the most prevalent automotive bus, CAN, is unencrypted and unauthenticated. This makes an adversary with access to the internal network even more powerful. Namely, ECUs expose a diagnostic interface which provides an external device connected over the CAN bus with powerful functionality such as read and write access to the internal memory. Both Keyword Protocol 2000 (KWP) [3] and its successor Unified Diagnostic Services (UDS) [2] standardise this diagnostic interface, and specify a security protocol to prevent unauthorised access to security sensitive functionality. However, the standards leave the choice of the cipher and key size up to the manufacturer. An adequate choice for this cipher is crucial for sufficiently securing this powerful interface.

Historically, the automotive industry has relied on proprietary cryptography to secure the critical parts of the network. Unsurprisingly, it has experienced a myriad of attacks on these proprietary cryptographic primitives (see e.g., [22, 70, 125]). For instance, manufacturers have a track record of using insecure, proprietary ciphers in immobiliser systems, the main theft-deterrent system in a car, since 2004. The immobiliser prevents so-called hot-wiring of a vehicle by cryptographically authenticating the transponder embedded in the keyfob. These flaws, typically exploitable with cheap, off-the-shelf equipment, stand in stark contrast with the often large purchase and upkeep cost of a vehicle. Manufacturers constantly migrate to newer, more secure ciphers with bigger internal states and key sizes. However, the transponder configuration and key derivation scheme used equally contribute to the security of the system. One weak link compromises the whole system and allows an attacker to bypass the immobiliser.

Due to their diverse hardware compositions and many peripherals, embedded devices are considered hard to analyse [130]. On top of that, automotive components typically run on esoteric architectures, which state-of-the-art analysis tools often do not support yet. This creates a three-pronged challenge: (*i*) firmware acquisition is non-trivial due to the lack of common debug interfaces. However, A microcontroller typically embeds an on-chip bootloader which is the first program to execute on startup, and usually incorporates a mechanism to protect the various on-chip memories against unauthorised access. This protection mechanism is crucial to prevent attackers from obtaining or even compromising the firmware. Fault injection attacks can specifically target the instruction checking the readout protection or the fuses containing the protection byte to bypass this mechanism. (*ii*) dynamic analysis, already troublesome on embedded devices with a more common architecture (e.g., ARM), is near impossible without expensive hardware and specialised (proprietary) software. Dynamic analysis techniques execute the firmware and typically depend on the ability to single step the device or emulate the firwmare, both complicated by peripherals and real-time requirements in ECUs. Furthermore, it typically requires a substantial amount of manual analysis to get the dynamic analysis in a feasible state. (*iii*) static analysis techniques, which are known to under approximate the analysed firmware, e.g., due to dynamically calculated targets, often rely on architecture specific call idioms to obtain a more complete view of the program. Furthermore, inferring the regions for a specific functionality is non-trivial due to the lack of symbols and debug strings. These challenges all add to the difficulty of creating an open and ultimately more secure automotive ecosystem.

Following Kerckhoff's principle [102], security should come by design and not through obscurity. The safeguarding of intellectual property by the manufacturers should not come at the cost of diminished security. The automotive community would clearly benefit from a systematic way to analyse automotive firmware and thereby raise the bar in automotive security. Therefore, this thesis concerns itself with the assessment of the current standard in automotive security. With this research, we intend to shift the current state-of-the-art in commodity embedded devices to the automotive realm, and ultimately make automotive research more accessible to the security community.

## 1.1 Contributions

The research presented in this thesis advances the state of the art in automotive security, more specifically in firmware analysis and acquisition.

### 1.1.1 Breaking Diagnostics: Firmware Extraction over CAN

We provide a security analysis of diagnostic protocols in ECU firmware: firstly, we reverse engineer the ciphers used in the diagnostic access control mechanism, which relies on a challenge response protocol, from ECU firmware by three manufacturers and present them in full detail. We identify flaws in each of them and present practical attacks to bypass all three with negligible computational complexity. Furthermore, we propose an online attack over CAN to recover the cryptographic secret without requiring any challenge response pairs. Secondly, we present a generic way to download and execute code on ECUs once we have bypassed the diagnostic authentication with our proposed attacks. This gives us read/write access to the internal memory of the ECU and its peripherals. We propose a firmware readout and modification framework on top of these features. The content presented in this chapter is based on the following publication:

*Van den Herrewegen J., Garcia F.D. (2018) Beneath the Bonnet: A Breakdown of Diagnostic Security. In: Lopez J., Zhou J., Soriano M. (eds) Computer Security. ESORICS 2018. Lecture Notes in Computer Science, vol 11098. Springer, Cham.*

### 1.1.2 Obtaining Firmware through Enhanced Embedded Bootloader Exploits

We present a security analysis of embedded bootloaders of three different chips. We read out the bootloader binary from each chip and analyse how they enforce the Code Readout Protection (CRP). We demonstrate how software exploits such as ROP are a threat even to embedded bootloaders of simple microcontrollers. We use dynamic analysis to enable a ROP exploit on the LPC bootloader. Next, we apply several static and dynamic analysis techniques to direct hardware-based fault-injection and bypass CRP mechanisms in the bootloaders of the STM8 and 78K0. We present the first double-glitch attack in full detail to bypass the CRP on two different STM8 chips. Next, we apply symbolic execution to the 78K0 bootloader binary to predict and classify glitch offsets. Finally, we systemise the vulnerable design patterns leading to the vulnerabilities presented in this chapter and in the wider literature.

Specifically, my contributions in the content presented in this chapter consist of (*i*) bringing static and dynamic analysis techniques to the hardware security realm to aid voltage glitching of the STM8 and 78K0 chips. (*ii*) systemising bootloader *anti-patterns*, which should be avoided when designing a bootloader. (*iii*) Development and open-sourcing of the code (available at `https://github.com/janvdherrewegen/bootl-attacks`) used to control the glitch hardware and perform the attacks. This chapter is based on the following publication:

*Van den Herrewegen, J., Oswald, D., Garcia, F. D., & Temeiza, Q. (2021). Fill your Boots: Enhanced Embedded Bootloader Exploits via Fault Injection and Binary Analysis. IACR Transactions on Cryptographic Hardware and Embedded Systems, 56-81.*

### 1.1.3 Security Analysis of DST80 Immobiliser ECUs

We present an end-to-end security analysis of two immobiliser systems built upon the proprietary DST80 cipher. Firstly, by improving on existing fault attacks we read out the immobiliser firmware of two manufacturers, from which we extract and reverse engineer the DST80 cipher, which we present in full detail. Furthermore, we expose key derivation weaknesses from the immobiliser firmware for both manufacturers, resulting in attacks with negligible computational complexity. Specifically, we show how the cryptographic keys in transponders by the first manufacturer, which we recovered from the immobiliser EEPROM, only have 24 bits of entropy. The second manufacturer uses a proprietary key derivation scheme, which we reverse engineered from the immobiliser firmware, to generate the cryptographic keys based on publicly readable transponder pages and a secret value in the firmware. Finally, we show how improperly configured transponders can lead to a full compromise of the cryptographic key.

My contributions in this chapter consist of (*i*) end-to-end analysis of two real-world DST80 immobiliser systems, (*ii*) reverse engineering and publishing of the DST80 cipher, (*iii*) exposing the transponder key diversification issues in both systems. This chapter is based on the following publication:

*Wouters, L., Van den Herrewegen, J., Garcia, F. D., Oswald, D., Gierlichs, B., & Preneel, B. (2020). Dismantling DST80-based Immobiliser Systems. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2020(2), 99-127.*

# Part I

# Background & Related Work

CHAPTER 2

# BACKGROUND

This thesis concerns itself with automotive and embedded firmware extraction and analysis techniques. In this chapter, we present the relevant technical background to our research. Firstly, we introduce several automotive standards and protocols which we will refer to throughout this thesis. Then, we introduce the reader to general embedded firmware extraction techniques and point out the difficulties of automotive firmware extraction specifically. Then, we detail our fault injection hardware setup we use in various attacks throughout this thesis. We outline the general functionality of a vehicle immobiliser system and introduce the DST80 transponder. Finally, we introduce the reader to the particularities of embedded firmware analysis.

## 2.1 Automotive Protocol Stack

Modern vehicles consist of dozens of interconnected ECUs, each responsible for a subset of its functionality. The industry has standardised various protocols and interfaces found on these ECUs, from the physical to the application layer. What follows is by no means an exhaustive list of all automotive protocols but is meant to introduce the reader to various building blocks of an automotive network mentioned throughout this thesis.

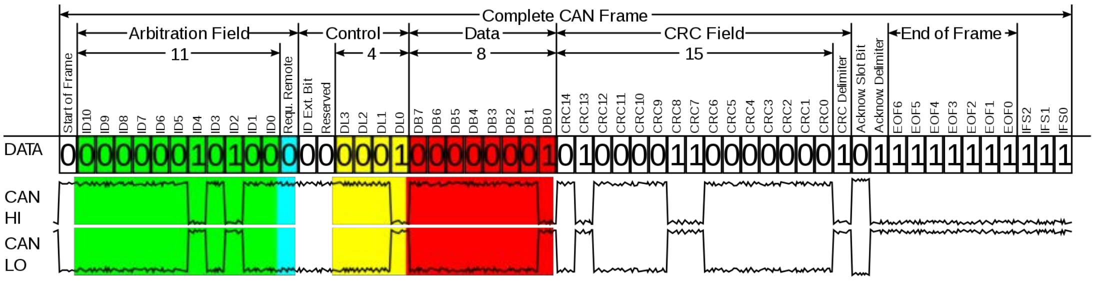


Figure 2.1: Structure and signals of a CAN frame[1]

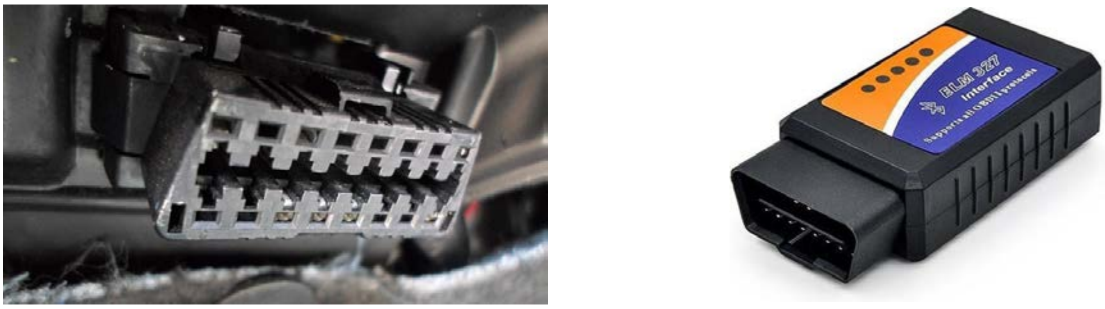
### 2.1.1 Physical & Data-Link Layer

On the physical and data-link layer, several vehicular buses interconnect various components ranging from ECUs to sensors and actuators. Typically, a transceiver separates the physical bus from the controller, which in turn forwards the received data to the µC and performs link-layer tasks such as validating a message CRC.

#### Controller Area Network

CAN, an automotive bus standardised in [1], is a differential serial bus with two signals: CANH and CANL. Originally designed by Robert Bosch Gmbh [23] for its robustness and reliability, the bus has a dominant level (where CANH is higher than CANL) indicating a 0 bit, and a recessive level indicating a 1. If two nodes transmit a message at the same time, the node which first deviates with a recessive bit loses the bus arbitration. Hence, as depicted in Figure 2.1 the first part of each CAN frame, the CANID, decides the arbitration on the bus. By design, frames with identifier `000` have the highest priority. The specification defines a standard format which uses 11 bit identifiers, which range from `000-7FF`, and an extended format with 29 bit identifiers. CAN has a publish-and-subscribe architecture: any node connected to the bus observes and acknowledges all traffic and may transmit on one or several identifiers. Furthermore, it supports bit rates up to 1Mbit/s and is by design unencrypted and unauthenticated. Thus, it relies on upper layer protocols to introduce security primitives if needed.

[1] CC BY-SA 3.0, `https://commons.wikimedia.org/w/index.php?curid=31571749`



(a) A typical 16-pin OBD-II connector in a vehicle[1]

(b) The ELM327, a typical diagnostic tool which connects to the OBD-II port.

Figure 2.2: A typical OBD-II port and a simple diagnostic device purposed to connect to it.

#### OBD-II

Every vehicle commissioned in the European Union since 2004 [54] is legally mandated to be equipped with an OBD-II port. As depicted in Figure 2.2a, it features a standardised 16-pin connector typically located below the driver's dashboard, through which a mechanic can query diagnostic trouble codes on several diagnostic buses (e.g., CAN). Diagnostic tools such as the one depicted in Figure 2.2b connect to the OBD-II port and incorporate drivers for the various OBD communication protocols used by different manufacturers. These devices usually expose a command interface (e.g., WiFi, Bluetooth or USB) which a generic phone or laptop can connect to.

### 2.1.2 Transport & Network Layer

Due to the limited message size (e.g., a standard CAN frame only contains 8 data bytes), nodes often use a transport layer protocol to encapsulate individual frames into bigger messages. We list two transport protocols we have observed in diagnostic communication, one standardised by International Organization for Standardization (ISO) and one reverse engineered from Volkswagen Auto Group (VAG) ECU firmware. However, we would like to emphasise that Original Equipment Manufacturers (OEMs) are by no means obligated to use these in regular ECU communication and often design their own protocol.

[1] Image Credit: Alain Van den Hende/Wikimedia Commons

### ISO Transport Layer (ISO-TP)

ISO 15765-2 specifies a transport layer on top of CAN [4]. It is specifically designed for diagnostic communication and specifies several frames which support messages up to 4095 byte. The standard defines the following frame types, indicated by the first 4 bit of each frame:

**`Single Frame`** Indicates a single message with up to 7 byte of data.

**`First Frame`** Specifies the total message length (12 bit) and contains the first 6 byte of data

**`Consecutive Frame`** Holds a frame index and the next 7 byte of message data.

**`Flow Control Frame`** Used for a node to acknowledge it has successfully processed all previous message data and is ready to receive new frames.

### Transport Protocol 2.0 (TP 2.0)

VAG has designed its own transport layer protocol, TP 2.0. Even though its specification is not publicly available, is has been described and implemented online [221]. Through these sources and our own reverse engineering efforts of multiple VAG ECUs (cf. Section 4.3.4), we briefly summarise the protocol here. Firstly, the client sets up a diagnostic channel with the ECU by sending a `Channel Setup` frame containing the two CANIDs which will be used for the diagnostic communication. Once the communication channel is set up, both nodes can transmit messages in a `Data Transmission` frame. The first nibble contains the opcode, which indicates the node's current status and simultaneously handles the flow control. While sending and receiving data, any node can either acknowledge previous data, or signal it is waiting for an acknowledgment. We refer the reader to [221] for a more detailed description of the protocol.

### 2.1.3 Application Layer

The automotive industry specifies several application layer diagnostic standards. Even though implementation details are often left up to the manufacturer, the standards define the high level communication between the diagnostic client and the ECU. Note that we use the concepts tester and client interchangeably, both denoting the diagnostic device querying the ECU.

#### Unified Diagnostic Services

UDS, standardised by ISO in [2], is the most prevalent diagnostic protocol on ECUs. The standard defines several diagnostic sessions: in the default session an ECU fulfills its normal functionality on the internal vehicular network. A diagnostic client can invoke the `DiagnosticSessionControl` service to change the active diagnostic session to either a programming or an extended diagnostic session. However, the available functionality in each of these is left up to the manufacturer. The main access control mechanism is a challenge-response (also known as `seed-key` in automotive terminology) protocol specified by the `SecurityAccess` service, as depicted in Figure 2.3. In order to authenticate to the ECU, a diagnostic client must send a *challenge request* to the ECU, which subsequently replies with a randomly generated challenge (also called the *seed* in automotive terminology). Both the client and ECU calculate a *response* (also called the *key* in automotive terminology) from this challenge according to a manufacturer-specific cipher, based on a shared secret. The client is authenticated if it supplies the ECU with a valid response. Multiple security levels are defined in UDS, which the manufacturer is free to use for different levels of access. UDS only specifies the challenge-response protocol, leaving the key size and choice of the cipher up to the manufacturer. The `RoutineControl` service executes preprogrammed functions in the ECU, with each routine uniquely defined by a two byte identifier. The client can pass arguments in a routine control call if needed. The standard specifies some routines and their respective identifiers, such as the `EraseMemory` routine with identifier `FF00`, while the identifier range `0200-C000` is reserved

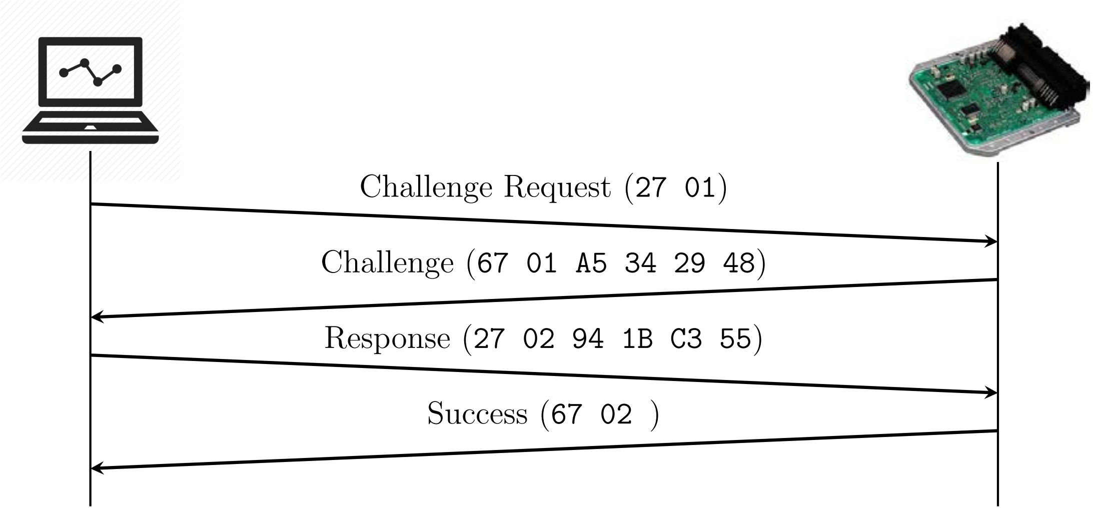

Challenge Request (27 01)
Challenge (67 01 A5 34 29 48)
Response (27 02 94 1B C3 55)
Success (67 02 )

Figure 2.3: The challenge-response protocol specified by UDS

for manufacturer specific use. Finally, the `RequestDownload` service provides a diagnostic client with functionality to download data to the ECU. Before sending the data with the `TransferData` service, the tester must specify an address where it will download the data to along with the size of the data. The tester should invoke the `RequestTransferExit` service on completion of the transfer.

### Keyword Protocol 2000

KWP is the predecessor of UDS. Both protocols are very similar, so will only outline the main features here. KWP works with `responsewords`, which generally coincide with service identifiers in UDS. The standard proposes the protocol on top of the K-line [93], an automotive serial bus designed for diagnostic services. In order not to impede with regular messages, it is used exclusively for diagnostic traffic. However, since KWP is an application protocol just like UDS, we have observed it both on the K-line as on the CAN bus.

### Universal Measurement and Calibration Protocol

Both Universal Measurement and Calibration Protocol (XCP) and its predecessor, Can Calibration Protocol (CCP) [107] are standardized by the Association for Standardization of Automation and Measuring Systems (ASAM). XCP is an application protocol which defines advanced features such as arbitrary read/write access to variables in ECU memory, synchronous data acquisition and flash programming of ECUs for development purposes. A diagnostic tool, also called the `master` in XCP, can analyse the connected ECUs, or `slaves`, through various XCP commands specified in the standard. The master has access to variables in memory by way of an ECU description file exclusive to each ECU, and can even download a reflashing kernel to the RAM for reprogramming purposes. Automotive software companies such as Vector Informatik support XCP solutions for ECUs of over 30 major automotive manufacturers [214], alluding to the extensive use of the XCP standard in the automotive industry.

#### Diagnostic communication channels

The application level protocols described here are independent from the physical and transport layer. Hence, neither UDS nor KWP specify the exact nature of the diagnostic communication channel (*i.e.*, for diagnostic communication over CAN, on which CAN ID each ECU listens for and responds to diagnostic messages). This is manufacturer specific, although there are some similarities across manufacturers. Since CAN frames with a lower identifier have priority over those with a higher identifier on the bus, diagnostic CAN identifiers are usually within the range `0x700-0x7FF`. Additionally, there is generally a clear relation between the CAN ID on which an ECU receives diagnostic messages, and on which ID it replies (e.g., $\mathrm{ID}_{send} = \mathrm{ID}_{recv} + 8$).

## 2.2 Automotive Supply Chain

As to be expected given the complexity of modern vehicles, car manufacturers do not produce every single piece of equipment in house. The automotive supply chain comprises a plethora of businesses and is roughly divided into the following levels:

**Original Equipment Manufacturers (OEM):** OEMs produce cars, which may contain a number of components by lower tier suppliers, in their own manufacturing plants. Examples of OEMs mentioned in this thesis are Ford, Volvo, Fiat (cf. chapter 4), Toyota and Kia/Hyundai (cf. chapter 6).

**Tier-1 suppliers:** first-tier suppliers, such as Bosch or Continental, have a direct relationship with the OEMs, and typically supply complete modules (e.g., the powertrain) to the manufacturer.

**Tier-2 suppliers:** second-tier suppliers do not interact with the OEM directly, but provide parts (e.g., a radar) to Tier-1 suppliers.

Various sections in this thesis assess the systems currently used in cars of various manufacturers. Typically, we carry out the research on ECUs purchased online (e.g., from scrapyards or eBay). The limited available information on the supply chain of a particular module further complicates pinpointing the responsible party for any found vulnerabilities. Hence, in this thesis, we assess where the flaw or vulnerability originates from based on our experiments. That is, if the same vulnerability is present on ECUs from different Tier-1 suppliers but from the same OEM, we will assume the OEM is responsible. Furthermore, due to the varied automotive landscape and the practical inability to assess every single car or model individually, it is not trivial to assess the impact of the presented vulnerabilities. However, where possible we include an estimate of the market share impacted based on freely available information.

## 2.3 Acquiring Automotive Firmware

Embedded devices and specifically automotive ECUs come in many shapes and forms. For that reason, there is no single clean-cut way of extracting firmware from ECUs. Unlike for Internet-of-Things (IoT) devices, where the firmware is often available on the merchant's website, automotive vendors do not typically provide such a service. Furthermore, automotive grade chips are specifically designed to operate and last in a challenging operating environment. They must function within a wide range of temperatures and due to their low-power requirements they commonly do not run on commodity architectures such as ARM. In this section, we outline several methods to acquire firmware from these often exotic automotive architectures.

### 2.3.1 Non-intrusive Firmware Acquisition

Recovering ECU firmware does not always require physical access to the actual ECU or hardware. This method is the least intrusive and is valuable in the scenario where we are interested in certain (proprietary) algorithms or protocols embedded on the ECU. Even though the OEM does not typically openly provide firmware for various reasons, automotive forums such as MHH Auto [123] and Digital Kaos [53] contain a considerable amount of automotive firmware of various makes and manufacturers. Furthermore, diagnostic tools can contain firmware updates for critical ECUs such as the Engine Control Unit or Body Control Module. We can then obtain the automotive firmware by extracting it from the diagnostic software.

### 2.3.2 Hardware-based Extraction

Online acquisition methods are typically effective for firmware of high demand ECUs such as the Engine Control Unit. However, they are less likely to succeed for niche components such as the immobiliser or keyless entry ECUs. Moreover, quite often we want to extract the firmware of a certain component, for instance to obtain cryptographic secrets specific to a range of ECU (e.g., a diagnostic key per model) or even to a single module (e.g., the immobiliser key). In these cases, we have no choice but to resort to the original hardware and lift the firmware from the device. Unlike online acquisition, these methods are often invasive and require dismantling parts of the ECU and locating and soldering several test points. Automotive µCs typically store the ECU firmware in internal flash memory, with ECU-specific data often stored in either internal data flash or external EEPROM. Common embedded firmware extraction techniques for Linux-based

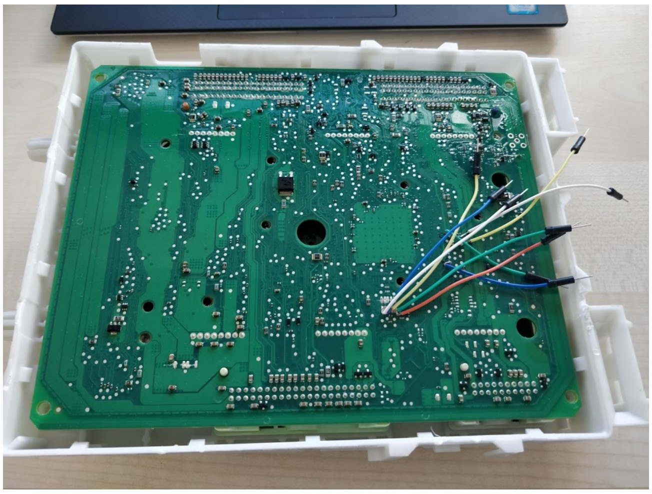


Figure 2.4: A JTAG interface on a Ford BCM incorporating a SPC560B microcontroller.

systems such as reading out the flash content over a shell through a serial interface [213] (e.g., Universal Asynchronous Receiver Transmitter (UART) or SPI) are not suitable for automotive components due to the lack of a common operating system. Namely, due to their real-time constraints, ECUs often run a bare-metal interrupt-driven firmware. Furthermore, even though many automotive-grade chips do incorporate a common JTAG interface (cf. Figure 2.4), open-source tools such as OpenOCD [153] do not provide support for all architectures or chip variants. After all, due to security concerns and to prevent competitors gaining insight into their products, many chip manufacturers do not release their JTAG specification to the public. However, low-cost (~$300) debugging equipment, such as the PEMicro Multilink [134] provides support for a range of automotive chips.

Other than JTAG, just like many other embedded devices, automotive μC typically expose one or several debug interfaces. Table 2.1 gives a non-exhaustive list of the debug interfaces we have encountered on chips of several automotive vendors. However, access to the debug interface does not automatically infer read access to the internal firmware. Typically the chip enforces a copy protection mechanism which prevents unauthorised memory access. By specifically targeting the CRP check, fault injection techniques can bypass this protection mechanism and gain access to the on-chip memory.

| Chip manuf. | Protocol | Interface(s) |
|---|---|---|
| Renesas | Renesas Flash Programming [166]<br>On-Chip Debug [43] | SPI, UART<br>JTAG |
| STMicro | ST Bootloader [191]<br>Single Wire Interface Module (SWIM) [192]<br>Serial-Wire Debug (SWD) [190]<br>On-Chip Debug [190] | SPI, UART<br>1-wire (DATA)<br>2-wire (SWDIO & SWCLK)<br>JTAG |
| NXP | Background Debug Mode (BDM) [146]<br>Nexus 2+ / JTAG [147] | 1-wire (BKGD)<br>JTAG |

Table 2.1: Non-exhaustive list of common automotive chips and their debug interfaces.

### 2.3.3 Fault Injection

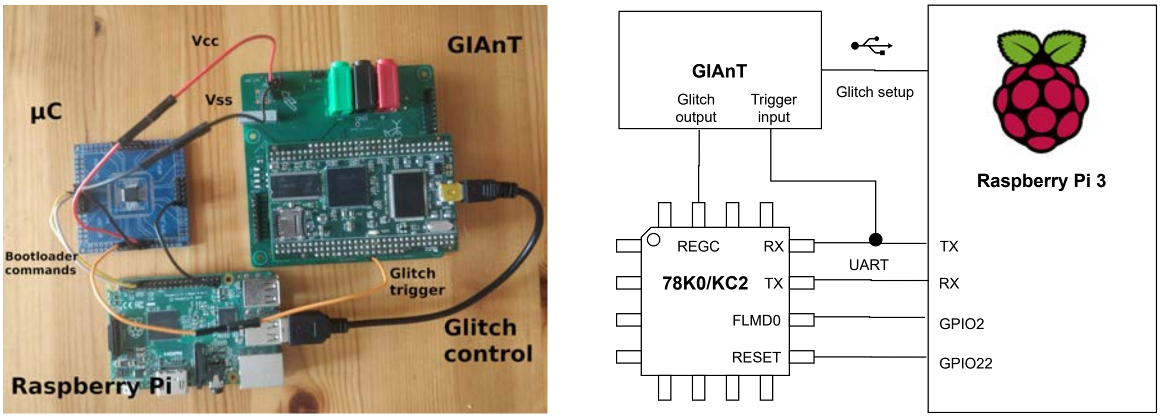
Voltage-Fault Injection (V-FI) is a fault injection technique which induces a disturbance on the target chip's power supply line with the aim to cause a faulty execution of a program. It can precisely target a single instruction, for instance when the bootloader evaluates the readout protection byte. We describe the hardware setup we use in the V-FI attacks presented in this thesis in detail and introduce notation to describe glitch parameters.

#### Hardware setup

For the voltage glitch attacks presented throughout this thesis we use a modified version of the Generic Implementation ANalysis Toolkit (GIAnT) [154] for generating the glitch waveforms. A Raspberry Pi 3 interfaces with the on-chip bootloader of the target μC over a serial interface (e.g., UART or SPI). The Raspberry Pi also sends the glitch parameters to the GIAnT over USB, which in turn introduces a glitch on the chip's $V_{CC}$.

#### Glitch parameters

Figure 2.6 illustrates the glitch parameters that we consider in this thesis together with our notation conventions. We refer to the normal operating voltage as $V_{CC}$, and the voltage of the glitch as $V_F$. $T_0$ denotes the offset in time of the first glitch from the trigger



(a) Hardware setup for the voltage glitching attacks.

(b) Schematic for our voltage glitching setup on a Renesas 78K0/KC2 μC.

Figure 2.5: Our hardware setup for the voltage glitching attacks described throughout this Thesis.

point (e.g., chip reset or a command sent over the communication interface), while $W_0$ is the width of the first glitch. In case of multi-glitch attacks, $T_1$ is then the offset of the second glitch from the end of the first glitch, $W_1$ the width of the second glitch, etc. In case of attacks that use only a single glitch, we omit the indices and simply use $T$ and $W$ for offset and width, respectively. Note that due to limitations of the GIAnT, the time resolution of width and offset is 10 ns.

## 2.4 Vehicle Theft Prevention

Throughout the years, the automotive industry has introduced several theft-deterrent systems. Before the introduction of electronic anti-theft devices, a criminal could simply short the car's ignition wire with the 12 V line (so-called hot-wiring) and start the vehicle. Hence, as early as 1972, vehicles started featuring a steering column lock, the first known vehicle anti-theft device [120]. Later, manufacturers introduced the immobiliser system, the first electronic anti-theft device.

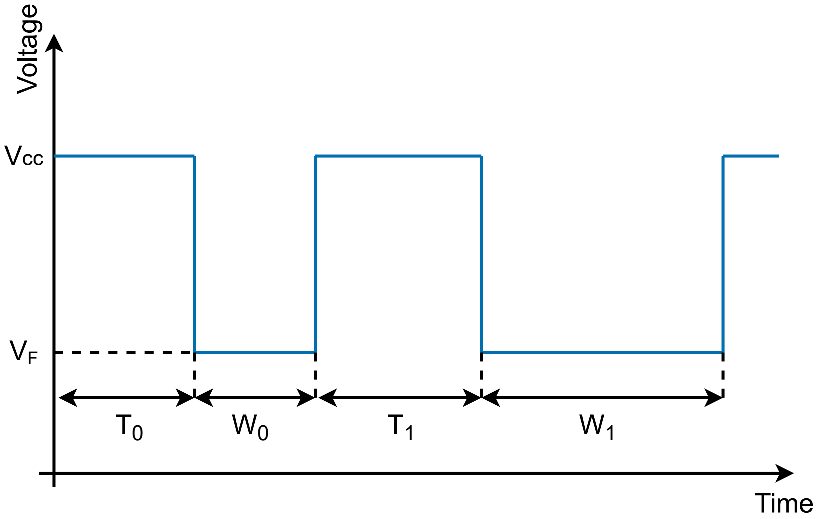


Figure 2.6: Parameter conventions for voltage glitches.

### 2.4.1 Immobiliser System

Its primary purpose is to prevent unauthorised use of the vehicle, which it achieves by requiring an RFID transponder to authenticate before starting the vehicle. Figure 2.7 depicts the general layout of such an immobiliser system, comprising an ignition coil, the immobiliser ECU, the Engine Control Unit and a transponder embedded in a key fob. The immobiliser identifies the transponder, which it powers through the ignition coil, through a Transponder Base-Station IC such as the TMS3705 [201]. Subsequently, it challenges the transponder chip, which authenticates by generating a cryptographic response. Only when this response is correct, the immobiliser authenticates to the Engine Control Unit in similar fashion to [20] and enables it to start the engine.

**Adversarial model** The immobiliser typically communicates over a Low Frequency (LF) interface with the transponder chip. This interface is unencrypted, and thus the immobiliser system stands or falls by the security of the underlying cryptographic primitives. An attacker in close proximity (*i.e.*, within several meters) could eavesdrop the communication, or worse, emulate the transponder with their own Radio Frequency IDentification (RFID) enabled device. Garcia et al. give an overview of the capabilities of such

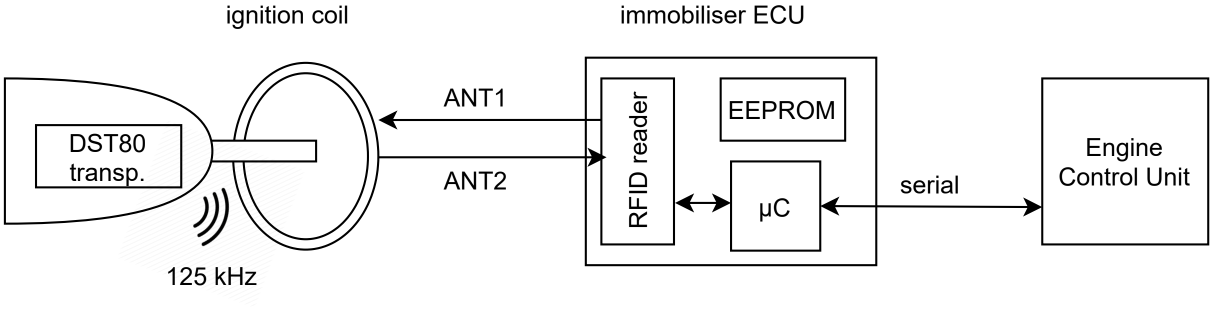


Figure 2.7: A typical immobiliser system

a device, the Proxmark, in [67].

### 2.4.2 DST80 Transponders

The Texas Instruments DST transponders are available in different package types; the classical wedge type transponder (TMS37145 [200]), a TSSOP package (TMS37126 [202]) and a TSSOP package containing both the TMS37126 and an MSP430 microcontroller (TMS37F128 [203]). All of these incorporate an 80-bit authentication key, which is internally stored in two 40-bit chunks, $key_L$ and $key_R$. The transponders can be configured either for fast authentication (the reader is not authenticated) or mutual authentication. DST80 transponders are uniquely identified by a 32-bit serial number stored in EEPROM consisting of a 3-byte unique identifier and a 1-byte manufacturer code. According to the datasheet, pages $1, 2$ and $3$ are always publicly readable. Additionally, the transponder incorporates several pages of EEPROM user memory, all of which can be write-locked. Table 2.2 gives an overview of several significant EEPROM pages of a DST80 transponder [200].

#### DST Authentication Scheme

DST80, just like DST40, builds upon the DST challenge-response protocol. Namely, the immobiliser randomly generates a 40 bit challenge and sends it to the transponder, which in turn calculates a 24 bit signature based on a shared secret and the underlying cipher

Table 2.2: DST80 transponder memory layout

| Page | Size (bytes) | Description |
|---|---|---|
| 1 | 1 | Selective addressing[1] |
| 2 | 1 | User data[2] |
| 3 | 4 | Transponder ID |
| 4 | 10 | Encryption key |
| 30 | 5 | Configuration page |

[1] Page 1 is also referred to as the `password` page. For example, in transponders used with Toyota immobilisers, it indicates whether the key is a master key or valet key.

[2] According to [200], this page contains the key number in the application. In Toyota transponders, it is called the `identity` page.

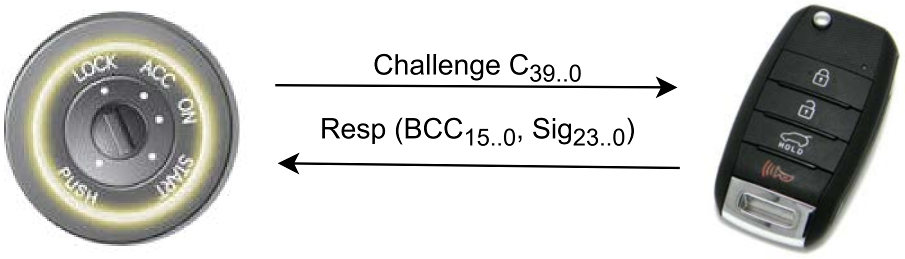


Figure 2.8: The DST authentication protocol

(*i.e.*, DST40, DST80 or DST-AES in the most recent transponders).

## 2.5 Firmware Analysis Techniques

In this section, we introduce the terminology and layout of embedded device firmware. We detail their composition, the approach taken to analyse them, and the specific challenges faced in each stage of the analysis process. Hereafter, we refer to firmware without an Operating System (OS) as *bare metal* and refer to the combination of a device and its firmware as a System Under Test (SUT).

**Interrupt-driven software** The execution of interrupt-driven software is directed by handling and processing interrupts. Interrupts are triggered by device peripherals (on- and off-chip), the OS, and system tasks. A task represents the execution of a part of software or firmware that is responsible for performing a specific function, e.g., input processing. Interrupt-driven software is composed of one or many tasks, which together constitute its functionality.

**Real-Time Operating System (RTOS)** A Real-Time Operating System (RTOS) is a specific type of interrupt-driven software that operates under soft or hard real-time constraints. Time-sensitive tasks orchestrated by the OS must be completed within strict windows to adhere to these constraints. A RTOS manages the execution of many concurrent tasks, where each task can be interrupted by one running at a higher priority, and interrupts can be nested.

### 2.5.1 Reverse Engineering Embedded Firmware

Reverse engineering is the process of taking a SUT and determining properties pertaining to its inner workings. This process necessarily involves a high degree of human intervention—both during the analysis stage and afterwards. This is because the objective of reverse engineering is to gain an understanding of the SUT such that, as a human analyst, we can explain how it works and extract specific implementation details.

Reverse engineering is performed using two complementary classes of analyses: static and dynamic. Static analyses form conclusions about a SUT by analysing it at rest, while dynamic analyses draw conclusions from observations of runtime behaviour. Interactive disassemblers such as IDA Pro [88] and Ghidra [209] are at the core of the majority of reverse engineering approaches. They enable an analyst to interact and manipulate a static representation of a firmware to gain a deep understanding of its functionality. While interactive disassemblers are the *de facto* tool for static reverse engineering, debuggers and execution tracers fulfil that role for dynamic approaches, allowing us to observe properties of the SUT's execution.

**Program database** Interactive disassemblers represent programs using a database. This database typically consists of the disassembly of each executable segment of the software, a computed control-flow graph, identified functions, static data references, and cross-references between code and data. In most cases, the database will be automatically populated by the tool to provide an initial representation of the program. Then, throughout the reverse engineering process, it will be manually refined, by adding discovered functions, control-flow edges, and data-type information.

**Firmware composition** RTOS-based and bare metal firmware do not have a common container format and vary widely in their composition. The only commonality is that their application logic, library code, static data, and OS (if present) all exist within the same binary blob. Thus, when analysing them with an interactive disassembler, manual intervention is often required to import them and trigger initial "automatic" analyses. That is, an analyst must manually define the memory mapping and Instruction Set Architecutre (ISA), and as interrupt-driven firmware does not have a single entry-point (e.g., `main`), they must must also identify interrupt vector tables, addresses of callbacks for tasks, and other initialisation routines. Moreover, while tools often include reasonable loading defaults for common chips, due to architectural variations these defaults are often inadequate.

**Firmware disassembly** Disassembly is the inverse process of assembly, *i.e.*, the translation of raw bytes into architecture specific instructions. A disassembler can only obtain correctly disassembled instructions if they are triggered from viable starting offsets within the firmware. Many instruction sets are variable length encoded (VLE) with short instruction widths and so have *valid* disassemblies at a large number of offsets, of which only one is correct. Further, since firmware often use non-standard and mixed calling conventions, common instruction patterns do not always give away these starting points. These problems are compounded by lack of mature support for embedded ISAs even in state of the art tools [66].

**Control-flow recovery** Control-flow recovery algorithms reconstruct a Control-Flow Graph (CFG) containing the transitions between blocks of disassembled instructions. A block in this case is a contiguous sequence of instructions terminated by a control-flow altering instruction, e.g., a branch or a call.

**Region inference** Embedded firmware intertwines application, OS and library code in one binary. Therefore, a reverse engineer must identify exactly which parts of the firmware to focus on. If present, symbols (e.g., function names) are very useful for this purpose. Furthermore, static data (e.g., debug strings) often gives a good idea of the functionality of the region or function it is located in. This static data can be located in close proximity to the corresponding functions, making them trivial to match. However, embedded firmware often stores static data in a separate segment, which it typically addresses indirectly (e.g., through the `gp (global pointer) for many Renesas µCs`. Therefore, when disassembly and control-flow recovery results in an incomplete program database, the amount of available data references to guide the reverse engineering process will be similarly limited.

**Debugging** Debugging involves pausing the execution of a SUT on specific conditions and observing its state (*i.e.*, registers and memory). For embedded software, the overhead caused by this invasive analysis technique can violate real-time constraints or interfere with timers, leading to an execution untrue to the target. Existing approaches (e.g., see [111, 131, 226]) modify the firmware in a way that ensures stability, leading to a device-in-the-loop approach. However, these modifications impact the faithfulness of analyses performed in an unquantifiable and unpredictable way. An additional challenge when analysing firmware in this way is correctly simulating its external environment, e.g., providing peripherals with the correct input. Without this, the firmware will *stall*, looping forever until it receives the correct input or is reset. Identifying stall points and handling them, either by stubbing or providing input is necessary to enable most kinds of dynamic analysis.

**Execution tracing** Execution tracing involves tracking part of the state of a SUT over the course of an execution. The granularity of this tracking is highly dependent on the tracing methodology used. Instrumentation-based tracing (see e.g., [51]), modifies the firmware to produce fine grained trace information. However, these approaches are often invasive and impact performance significantly. Many devices offer hardware-based support for execution tracing (see e.g., [10, 9]. In this case, additional on-chip hardware captures certain types of events—typically the value of the program counter on branches—and either outputs it over a hardware trace port or stores it in memory. In the latter case (which does not require expensive hardware), traces are inevitably limited in size. To accommodate this, tracing technologies typically offer configurations to start tracing on a particular program counter value, or only include certain regions in memory.

## 2.6 Notation and Variables

Throughout this thesis, we will maintain the following notation. $S_i$ denotes bit $i$ of a variable $S$, with $S_0$ being the least significant (rightmost) bit. Likewise, $S[i]$ stands for byte $i$ of $S$, with $S[0]$ the least significant (rightmost) byte. We depict a rotation of $S$ by $i$ bits to the left by $S \lll i$. $S_{i \dots j}$ characterises a range from bit $i$ to bit $j$ (including bits $i$ and $j$) of variable $S$. Finally, $(v, w)$ denotes a concatenation of bytes $v$ and $w$, with $w$

the least significant byte.

CHAPTER 3

# RELATED WORK

## Contents

In Chapter 2, we introduced various building blocks of a modern vehicle. Intuitively, securely integrating all of these standards, protocols and security primitives on top of a decades old (mechanical) design is not a trivial task. In this chapter, we introduce the literature that builds on top of these to make the automotive environment more secure. Furthermore, to illustrate the consequences of an insecure design or implementation, we list the existing work exploiting vulnerabilities in real-world systems. This includes local and remote attacks on the vehicular network and techniques to compromise individual ECUs. In addition to that, we give an overview of the literature on immobiliser and Remote Keyless Entry (RKE) security. We follow up with the related work regarding fault injection and its use in attacks on firmware readout protection. We conclude this chapter with an overview of the literature on embedded device analysis techniques.

# 3.1 Automotive Network and Component Security

## 3.1.1 Automotive Networks

The automotive network comprises multiple automotive buses, some more critical than others and each with their own approach to secure communication. Most standards were designed under the assumption of a closed vehicular network without adversaries and thus without security in mind. Unsurprisingly, the literature describes an array of attacks which assume an adversary has physical access to the bus.

**Physical & Data-Link** On the physical layer, Gessner et al. propose a physical fault-injection framework on CAN, which introduces erroneous bit sequences on the bus [72]. Additionally, CAN is no stranger to link-layer attacks: it is susceptible to a Denial-of-Service (DoS) attack by bit-banging, which causes a transmitting node to error [155, 38, 137]. If the bus signal does not correspond to the bit transmitted by a node, it assumes an error occurred and retransmits the frame. After 255 failed transmissions, the node enters a *Bus-off* mode, where it does not transmit any more frames until the node is reset. Likewise, flooding the bus with frames containing of highest priority ID (`000`) results in a trivial DoS attack [28, 63]. Murvay et al. prove that similar DoS attacks apply to the Flexray bus in [138]. They exploit the physical and data-link layer both to flood the bus and spoof messages.

**Authentication and Encryption** The design decision of both CAN and FlexRay buses to prioritise reliability and safety over security are at the root of these DoS attacks. However, in order to counter attacks requiring physical access to the CAN bus, the literature suggests several authentication and encryption schemes on top of CAN. Radu et al. propose LeiA [163], a light-weight authentication protocol for ECUs connected to the CAN bus. Each CAN ID has a corresponding authentication key, which a node uses to calculate a Message Authentication Code (MAC) on the message. Receiving nodes can check the validity of a message by comparing the calculated MAC to the MAC embedded in the 18 bit extended identifier field of the CAN frame. Several other protocols follow a similar pattern to provide message integrity to CAN by embedding a MAC in the frame [86, 219, 143, 16, 211, 79]. Moreover, Kurachi et al. propose caCAN in [113], which introduces a monitor node on the CAN bus that detects and destroys anomalous messages. In addition to message authentication, both LCAP [87] and TOUCAN [16] provide encryption using the RC4 and AES-128 cipher respectively. Finally, CANAuth [211] and LiBrA-CAN [79] provide authentication on top of the CAN+ protocol [228], a CAN compatible protocol which increases bandwidth by sending on otherwise unused timeslots. In similar fashion, Groza et al. propose an Intrusion Detection System (IDS) which uses vacant timing slots as a covert authentication channel on CAN [80].

**Intrusion Detection** Another approach to mitigate against the physical attacker on the CAN bus is to detect anomalous messages from a compromised node and destroy them if required. Research has shown that ECU fingerprinting is an effective technique to accomplish just that [108]. Choi et al. introduce two kinds of adversaries on the CAN bus [39]: (*i*) a *Type I* adversary gains access by adding an additional node to the bus, e.g., through the OBD-II port. (*ii*) In contrast, a *Type II* adversary has compromised an ECU already connected to the bus, which it can use to inject frames. The latter can only reveal themselves when injecting a frame with an identifier otherwise not used by the compromised node. Statistical analysis on the voltage of CAN signals has proven a successful technique to uniquely identify ECUs [136, 39]. In [39], Choi et al. statistically analyse the electrical signal of the extended identifier emitted by the transceiver to classify each ECU, while in [136] the authors analyse the 11 bit ID. Building on this, Choi et al. propose a vehicular IDS that distinguishes between a genuine CAN bit error and a *Bus-off* attack as described earlier [39]. Their approach introduces a monitoring node on the CAN bus to detect adversarial nodes. In contrast, Cho et al. propose a clock-based IDS, which detects anomalies in the otherwise periodic nature of CAN messages on the bus [38]. Finally, Murvay et al. fingerprint ECUs based on the propagation time of the signal and thus their physical location in the car in [139].

**Message Reverse Engineering** Due to the proprietary nature of CAN communication, a significant reverse engineering effort is required to recover the message structure. Namely, OEMs typically store the CAN message decoding rules in so called Database CAN (DBC) files, which are not publicly accessible. 8 byte messages can be split up into 1 bit fields of information, making reverse engineering the message formats an arduous task. Despite the requirement of a very good understanding of the internal vehicular network architecture, manually reverse engineering message formats of critical CAN messages has proven a successful approach [125, 126]. For instance, Miller et al. recovered the message formats on Ford and Toyota cars to spoof both the odometer and speedometer and even obtained limited steering capabilities [125]. However, automated approaches have significantly reduced the effort required to reverse engineer the bus messages. In [157], Pesé et al. correlate diagnostic data acquired through OBD-II, sensor data (e.g., 3D gyroscope and accelerometer data) from a smartphone and a raw CAN recording from a test drive to recover the format of the CAN messages. In contrast, Wen et al. remove the need for a physical car by extracting CAN commands from car companion mobile apps [220]. They recover the semantics of CAN commands through backward program slicing from the standardised network APIs (e.g., Bluetooth and WiFi) and forced execution of the CAN command generation in the apps.

### 3.1.2 ECU Security

Ideally, when an attacker compromises one ECU on the network, they do not gain access to all cricital functionality of the vehicle. A gateway separates critical high-speed CAN buses from other low-speed, buses purposed for entertainment. However, the success of this compartmentalisation relies on the individual ECU security. That is to say, if an attacker compromises an ECU which is connected to several buses, they can use it as a bridge and gain access to other connected components. Checkoway et al. classify a car's attack surface into four categories, depending on how an attacker would gain access to the internal vehicular network (e.g., the CAN bus) [36]:

**direct physical:** e.g., an attacker gains direct access to the vehicle's internal network, e.g., through the OBD-II port or a parking sensor connected to the CAN bus.

**indirect physical:** e.g., attacks on the infotainment system through USB, CD or ethernet.

**short-range wireless:** e.g., attacks over Bluetooth or WiFi, where the attacker must be in close proximity to the car to succeed. Attacks on the immobiliser or keyless entry system equally fall into this category.

**long-range wireless:** e.g., attacks on the Telematics Control Unit (TCU) over the cellular network or on the FM radio component in the car. The authors also consider broadcast channels such as GPS part of this category.

In this section, we categorise the existing literature into these classifications, based on the attack vector.

### Direct & Indirect Physical Attacks

Koscher et al. pioneered automotive security research with an experimental security study of ECUs [110]. Provided *physical access* to the vehicle's OBD-II port, they manipulated ECU behaviour by fuzzing with random CAN payloads and by directed use of diagnostic packets. The authors tamper with several safety-critical ECUs by sending diagnostic messages, while they reprogram the TCU to act as a bridge between the high-speed and low-speed CAN bus in the vehicle. The reprogramming of the unit consists of downloading code to its Random Access Memory (RAM) memory and executing it. The authors did not encounter any access control mechanisms on the studied ECUs. Next, Miller and Valasek reverse engineered the complete reprogramming procedure on two Ford ECUs by eavesdropping the communication between a diagnostic tool and the ECUs in [125]. The tool reprograms the components by downloading a piece of code, which subsequently handles the reflashing procedure, to the RAM of the ECU. The authors abuse this mechanism to execute their own code on the ECU. The authors use several diagnostic services to authenticate and download code to the ECU, after which they invoke another diagnostic routine to jump to the downloaded code. No access restrictions are in place on the µC, giving them full access to peripherals such as the CAN bus and thus proving an attacker with access to the CAN bus can take control of other connected ECUs. Similarly, Cai et al. perform a comprehensive security analysis of BMW cars in [30]. The authors gain access to the infotainment unit and TCU through various local and remote attack vectors such as the OBD-II port, USB and a cellular connection, from which they can inject arbitrary messages on the CAN bus and control the vehicle. With either a USB-to-ethernet adapter or the OBD-II port provide the authors access to a diagnosis service on the infotainment, which implements a custom UDS protocol. It supports the upload and execution of a digitally signed shell script. The authors circumvent the signature procedure through a time-of-check to time-of-use (TOCTOU) vulnerability. Both this attack and an additional buffer overflow vulnerability in the navigation update procedure grant the authors root access on the head unit.

Regarding diagnostic security, Foster et al. describe how an attacker may exploit various diagnostic services [63] . Additionally, Khan [104] raises several issues on security in the UDS specification, the access control mechanism in particular. The paper states multiple security flaws in the `security access` service provided by UDS, more specifically on challenge generation and the complexity of the employed cipher. Furthermore, Khan notes that an attacker can recover the cipher and secret keys from the firmware.

### Remote Attacks

Critics deem the assumption of physical access to the vehicle bus as unrealistic. However, with the introduction of common technology such as WiFi, Bluetooth, cellular and other remote interfaces in modern vehicles, the literature has described a multitude of remote attacks.

**Sensors** A Tire Pressure Monitoring System (TPMS) is an in-vehicle Radio Frequency (RF) sensor network purposed to continuously monitor the air pressure in all tires, mandated by EU law in all vehicles from 2012 [165]. Rouf et al. perform a study of these wireless TPMSs in [170]. By eavesdropping on the wireless communication between the sensor and the central ECU, they uncover several security and privacy issues. Due to the lack of any encryption in the wireless sensor communication, an attacker can easily spoof messages. Furthermore, an attacker can recognise and track a vehicle through the static 32 bit sensor IDs from a 40 m distance. Worse, the vehicle ECU does not appear to perform input validation and crashes upon reception of some invalid messages. Furthermore, [114] analyses the Tesla autopilot security, which is heavily dependent on the in-vehicle sensor network. Namely, the vehicle's automatic wiper function uses optical sensors and lane detection makes decisions based exclusively on camera data. A neural network decides whether to activate the wipers based on the light influx. The authors distort this by projecting an image containing noise, which activates the wipers without the presence of any rain. Likewise, the authors deceive the neural network in control of lane detection by making small physical alterations to lane markings.

**Bluetooth** Regarding short-range wireless attacks, Checkoway et al. pinpoint several exploitable calls to `strcpy` in the proprietary Bluetooth stack in the TCU, resulting in CAN injection capabilities [36]. Furhermore, in [199], Tencent Keen Security Lab exploit two vulnerabilities in the bluetooth stack to gain remote code execution on an AVN

(Audio, Visual and Navigation) Unit of a 2017 Lexus car. Since both vulnerable sections (a heap-based buffer overflow and an out-of-bounds heap memory read) are situated in code used to set up the connection before pairing, an attacker can exploit them seamlessly when they are in close proximity to the vehicle. Due to the lack of secure boot on the AVN, the attacker can then install a persistent exploit to fully compromise the module.

**WiFi** Many modern cars incorporate an in-vehicle WiFi hotspot, which a user can typically enable by inserting a data SIM or by paying for a manufacturer specific data subscription. In [141], Nie et al. investigate the security of the Tesla Model S. Through several vulnerabilities in the JavaScript engine of the web browser they gain a low-privilege shell on the head unit. An outdated Linux kernel reveals the presence of a well known vulnerability, which the authors exploit to gain a root shell. From there, the authors could trivially access the instrument cluster and Wifi module as root over SSH and telnet respectively without authenticating. Gaining access to the gateway proved more challenging: a specially crafted packet sent to an open UDP port opens a Telnet port, which consequently requires an authentication token. However, this token is static and embedded into the firmware of the gateway. Furthermore, the authors reverse engineer the update procedure of the gateway, which only performs a CRC32 check and thus is easily bypassed, and flash their own custom firmware. The diagnostic authentication on various ECUs consists of XORing a challenge with a constant, so the authors could trivially bypass this and alter behaviour through UDS messages. Moreover, in [198], Tencent Keen Security Lab investigate the Tesla S in-car WiFi module, nicknamed the Parrot. It incorporates a Marvell WiFi and Bluetooth System-on-Chip (SoC) containing a heap-based buffer overflow vulnerability, which is exploitable with a $\tilde{2}5\%$ success rate. Furthermore, once an attacker gains code execution on the WiFi SoC, a kernel based heap overflow exploitable over the SDIO (Secure Digital Input Output) interface hands them control over the Linux system running on the Parrot. In both instances, Tesla fixed all vulnerabilities immediately with an over-the-air update.

**Cellular** Checkoway et al. demonstrate the powerful capabilities of a remote attacker by exploiting vulnerabilities in the aqLink protocol used in the communication link between the TCU and a remote Telematic Control Center (TCC) [36]. They reverse engineer the protocol and exploit a buffer overflow in the low-level packet parsing code. Combined with a weak Random Number Generator (RNG) process and the use of static nonces this leads to a full remote exploit where an attacker can call into the modem and inject CAN frames. In similar fashion, Miller and Valasek gain remote access to the infotainment unit of a Jeep through an open TCP port exposing a Remote Procedure Call (RPC) interface, ultimately allowing them to inject arbitrary CAN messages on the vehicular network [126]. Worryingly, this TCP port is not only accessible through the in-vehicle WiFi, but also through the car's cellular connection. Worse, due to a network configuration error from the provider, any attacker on the same cellular network could access all vehicles with an enabled data connection. Finally, to form a bridge between the application processor exposing the RPC interface and the automotive μC (a Renesas V850 chip) with access to the CAN bus, the authors reflash the automotive chip with modified firmware, which forwards messages on to the CAN bus. More recently, vulnerabilities in the infotainment system have lead to the compromise of the internal vehicular network of Audi and VW cars [103]. Through a vulnerable service accessible via the WiFi hotspot, the authors have shell access to the infotainment unit of a VW car. Similar to [126], an attacker can access the same vulnerable service over the internet, given that they are on the same cellular network. Again, a V850 chip on the same Printed Circuit Board (PCB) handles the CAN communication. The authors show they can insert a backdoor on the V850 to allow arbitrary CAN message injection through a firmware update. Finally, the authors point out that the CAN bus gateway, which acts as a bridge/firewall for several CAN buses, only accepts signed firmware updates. In addition to the various local attack vectors described earlier, Cai et al. compromise a BMW head unit over a cellular connection [30]. They set up a fake base station to intercept cellular traffic from the car, which exposes an old WebKit client. The head unit uses this client, containing known vulnerabilities, to query online news updates. This leads, along with the local vulnerabilities, to a complete compromise of the head unit.

**Next Generation Telematics Protocol (NGTP)** NGTP is an over-the-air diagnosis protocol, based on the ASN.1 syntax, developed and adopted by several automotive manufacturers. Spaar reverse engineered the message formats from the TCU firmware and uncovers the use of DES and AES-128 static encryption keys [188]. The TCU encrypts and signs messages with one of 16 key pairs, chosen based on the message header. On reception of a simple SMS message, the car wakes up and sends an HTTP GET request, which is only encrypted and signed with the static NGTP keys. Thus, by replaying e.g., the door unlock message, an attacker can easily gain access to the car. Even when remote services are disabled on the vehicle, an attacker can simply send an activation message containing the Vehicle Identification Number (VIN) of the car, after which the original attack succeeds again. In the later BMW models covered in [30], the plain HTTP issue has been fixed and the TCU at least connects to the central server over HTTPS, mitigating the simple replay attack. However, the authors discovered that any NGTP message encapsulated in HTTPS can equally be sent over SMS. By sending an SMS which forces the car to connect to an attacker-controlled update server, the attacker can inject a payload which exploits a stack-based buffer overflow in the signature verification of the message. Then, using ROP the authors gain remote code execution on the baseband processor of the TCU. Finally, the authors complete the attack by exploiting the diagnostic role of the TCU to make the Gateway forward UDS messages to other ECUs on several CAN buses. The research points out there is no speed limit on the diagnostic communication, which could lead to e.g., a catastrophic reset of all ECUs while driving. To mitigate these attacks, the authors cooperated with BMW and suggested the exclusive use of HTTPS (instead of also allowing SMS) to verify whether the remote service request is coming from the BMW server.

**Aftermarket Devices** Drivers often add a so-called *aftermarket* device to the internal vehicular network via the OBD-II port for e.g., insurance purposes. In [36], Checkoway et al. describe several wireless configuration vulnerabilities in a so-called PassThru device, which connects to the OBD-II port, granting attackers a shell on the device and thus CAN injection capabilities. Likewise, Foster et al. exploit several local and remote vulnerabilities on an aftermarket device plugged into the OBD-II port to gain remote access to the vehicle [62]. An attacker can trivially access the device over unauthenticated SSH and Telnet services through either a USB connection showing up as a network interface or a cellular connection. Moreover, the authors reverse engineer the firmware update procedure, which triggers on receipt of an SMS containing the update server details. Arbitrary code execution ensues since the device does not authenticate the server and updates are not cryptographically signed. Once an attacker gains access to the device, they can inject arbitrary messages on the CAN bus.

## 3.2 Immobiliser and Keyless Entry Security

Vehicle theft has been an ongoing area of concern for automotive manufacturers. Once cars became prevalent in day-to-day life, so began the everlasting cat-and-mouse game of criminals coming up with new ways to steal vehicles and the manufacturers introducing innovative anti-theft measures in their newest range. In the very beginning, when automobiles were not the computerised appliances they are today, mechanical locks formed the main barrier of defense against criminals. A simple mechanical key granted access to the vehicle and unlocked the ignition to fire up the engine. Car owners could add a physical steering lock to further prevent vehicle theft. However, given physical access to or imagery of the key, purely mechanical keys are easy to replicate, nullifying the added security of mechanical locks. Hence, it was not until the early 1990s with the introduction of electronic immobilisers that vehicle theft began to decline [128]. Due to their effectiveness, they were mandated by law in the European Union in 1995 [42] and are an integral part of any motorised vehicle today. Along with the immobiliser, manufacturers started introducing keyless entry systems into their fleet in the 1980s. In 1978, Lipschutz laid the foundation for our current RKE systems with his remote control device for vehicle locks [117]. His system uses an infrared transmitter which communicates with a receiver mounted in the vehicle which operates the locks. Several iterations of this technology (e.g., replacing infrared with a more reliable High Frequency (HF) radio signal) lead us to the modern RKE systems. A typical modern car key incorporates a high-frequency radio transmitter for the Passive Keyless Entry (PKE) or RKE system alongside a low-frequency RFID immobiliser transponder.

### 3.2.1 Immobiliser Security

At first, immobilisers used fixed-code *read-only* transponders and thus did not provide any cryptographic security [98]. An attacker who could eavesdrop the transponder communication or wirelessly communicate with the transponder recovered the code and could easily authenticate to the car. Rolling codes were introduced to mitigate this problem, where transponders modify their code in EEPROM for each transaction. Then, with the introduction of Texas Instrument's DST in 1995 [98] and NXP's HITAG came the first cryptographically enabled transponders. The transponder authenticates to the in-vehicle reader using a challenge-response protocol based on a shared secret. Finally, transponders can implement a mutual authentication mechanism, where the reader must first provide the transponder with proof of the shared key before the transponder encrypts the challenge [98]. Because of their security critical function, immobilisers have been a hot topic of research ever since they became an integral part of the car's internal network.

**DST40** As early as 2005, Bono et al. published and cracked the DST40 cipher used in immobiliser systems of millions of vehicles and electronic payment systems [22]. DST40, based on the Kaiser cipher described in [97], is a block cipher underpinning the DST challenge-response protocol (cf. Section 2.4.2). Specifically, the reader generates a random 40-bit challenge and transmits it to the transponder. The transponder in turn encrypts the challenge with the shared key and sends the 40-bit response, consisting of a 16-bit Block Check Character (BCC) and a 24-bit signature (e.g., the least significant 24 bit of the encrypted challenge). DST40, the underlying cipher in Bono's case, incorporates a 40-bit challenge register along with the 40-bit key register. Each round, it inputs the challenge bits along with key bits into a Feistel network $F$ that generates a 2-bit output, which is subsequently XORed with the two leftmost bits of the challenge register. The key register is an Linear Feedback Shift Register (LFSR) which is updated every three rounds with taps on positions 18, 20, 37 and 39. The cipher runs for a total of 200 rounds. The authors built an FPGA cluster to exhaustively search the 40 bit key space and recover a DST40 key within an hour. More recently, Wouters et al. uncovered the use of the DST40 cipher in Passive Keyless Entry and Start (PKES) systems of several high-end contemporary supercars, including certain Tesla and McLaren models [222]. The cars use the DST authentication protocol in combination with DST40 both for the keyless entry and immobiliser systems. The authors reverse engineered the keyless entry protocol and propose a Time-Memory Tradeoff (TMTO) attack on DST40, requiring 2 challenge-response pairs and a 5.4 TB lookup table.

**DST80** With DST40 being proven insecure by Bono et al. in 2005 [22], Texas Instruments (TI) subsequently put forward its successor, DST80. As the name suggests, TI has doubled the key size of the cipher to 80 bit. Kammerstetter et al. propose an automated IC reverse engineering approach to recover DST80 from silicon in [99]. After an intricate process, which involves decapsultating the transponder, etching and polishing the chip and finally imaging the IC layers, the authors recover the netlist (e.g., the layout of the electronic connections) of the TMS37145 [200] immobiliser tag. They identify a 15 bit proprietary cpu core which can execute a specific DST80 instruction. However, they leave the reverse engineering and publication of DST80 as future work.

**Megamos Crypto** Verdult et al. propose several attacks on Megamos Crypto, another cipher widely used in car immobilisers in [216]. The authors reverse engineer and present the cipher, which uses a 96-bit secret key, in full detail. Furthermore, they propose a cryptanalytic attack, reducing the complexity from $2^{96}$ to $2^{49}$ encryptions. This attack, applicable regardless of the transponder configuration, exploits weaknesses in the cipher design such as a smaller 56-bit internal state (in relation to the 96-bit key), the lack of a random number generator, an invertible cipher state successor function and leakage of 15 bit of known-plaintext by the authentication protocol. Moreover, the transponder is often write-protected with a default PIN and the secret key is written in 16-bit chunks instead of an atomic 96-bit block. Hence, the authors recover a 96-bit key in 30 minutes and $3 \times 2^{16}$ encryptions. Finally, low entropy in the secret key (e.g., the first 32 key bit are all zero) leads to full recovery of the key in minutes, requiring only two authentication traces and a 1.5 TB rainbow table.

**AUT64** Moreover, Hicks et al. disclose several vulnerabilities in AUT64, first referred to in [70], a 64-bit Feistel network block cipher used in immobiliser systems of Mazda, Ford and Proton cars [89]. First, the authors identify certain weak keys, which make the cipher behave in an undesirable way. Then, they identify weaknesses in the compression function, which maps the 8 byte state to a single byte in the first step of the Feistel function, and in the substitution-permutation network. This leads to recovery of the 120-bit secret key with 512 plaintext-ciphertext pairs and $2^{37.3}$ offline encryptions for the eight round cipher implementation. The 24 round implementation requires 2 plaintext-ciphertext pairs and $2^{48.3}$ offline encryptions to recover the full key. Finally, the authors also identify weak keys in the implementation of a major vendor, reducing the entropy of the original 120-bit secret key to 59.5 bit.

**Immobiliser Authentication** As shown in Figure 2.7, the immobiliser and Engine Control Unit are typically not located within the same physical entity. The immobiliser authenticates the key fob, and if successful it relays this information to other ECUs in the car which then initiate the engine startup procedure. Thus, the security of the immobiliser system not only depends on the cipher choice, but also on the in-vehicle communication afterwards. In [20], Bokslag identifies and assesses the security of the cryptographic primitives used in the link between the immobiliser and the Engine Control Module in three different car makes. All three models use a challenge-response protocol, with only one implementing mutual authentication. The authors propose car-only attacks to recover secret key material for two of the models, while the third uses an obsolete transponder.

### 3.2.2 Keyless Entry Security

Before electronic car keys were commonplace, an attacker could simply lock-pick the door to gain access to the vehicle. Even with remote car keys, non-cryptographic attacks such as jamming the *lock* signal have proven successful. Moreover, the aim of a so-called *roll-jam* attack is two-fold: the attacker jams and eavesdrops the original unlock signal, thereby obtaining a valid rolling code, which they then use to gain access to the vehicle [224]. These jamming attacks only permit attackers entry to the car and do not allow them to start it. However, many immobiliser ciphers recur in keyless entry systems, making them an appealing target for cryptanalysis.

**KeeLoq** For instance, it has been demonstrated that the Keeloq cipher, used in various devices ranging from immobilisers and RKE systems to remote door locks, is vulnerable to a plethora of physical and cryptanalytical attacks [100, 19, 92, 48, 57]. Bogdanov et al. proposed the first attack on the cipher in [19], reducing the attack complexity from $2^{64}$ to $2^{37}$. They base their attack on weaknesses in the key schedule, which allows to set up a slide attack, and the cyclic structure of the cipher introduced in [48]. Indesteege et al. point out unrealistic memory latency assumptions of this attack in [92] and propose a practical attack on Keeloq which requires $2^{44.5}$ encryptions. The attack, based on original slide attacks and a novel meet-in-the-middle approach, requires roughly an hour of physical access to the key fob to acquire the traces and takes ~8 days of computation on 64 CPU cores to recover the 64 bit key. Furthermore, Eisenbarth et al. leverage side-channel cryptanalysis of the power consumption of hardware and software KeeLoq implementations to fully recover both the secret key from a transmitter and the master key from a receiver. They propose a differential power analysis which only requires 10 power traces and can recover the secret key in minutes [57]. However, the lack of a good trigger point in software implementations of the cipher brings along trace alignment problems, crucial for Differential Power Analysis (DPA). Moreover, the correlation coefficient of the correct key decreases as the number of rounds increases in a software implementation. Kasper et al. estimate a successful DPA attack extracting the master key of a software implementation requires a total of 10,000 traces instead of 10 . Improving on this, Kasper et al. recover a manufacturer key in a matter of seconds with only a single power trace by performing a Simple Power Analysis (SPA) of the KeeLoq decryption [100]. Timing differences in the round execution time of a reference implementation reveal one bit of the status register each round. Their technique automatically extracts the parameters required for the SPA from a single power trace and does not require any knowledge about the plaintext or ciphertext.

**Hitag2** In 2012, Verdult et al. found several flaws in the Hitag2 cipher, used in both immobilisers and RKE, allowing an attacker to bypass the immobiliser protection in dozens of car makes within minutes [215] using a Proxmark III [67] to emulate the transponder. They point out the lack of pseudo-random number generator in Hitag2 transponders, making them vulnerable to replay attacks. Furthermore, the cipher leaks one bit of information about the secret state every four authentication attemps on average. Finally, only 32 bit of the 48 bit internal state are randomly initialised, leaving 16 bit persistent over several authentications. These weaknesses lead to a full key recovery within seconds provided wireless communication with the transponder, and less than six minutes for a car-only attack. In [70], Garcia et al. revisit the Hitag2 cipher, this time tackling its use in RKE systems with rolling codes. They present a novel correlation-based attack in order to bypass the RKE system in vehicles of several manufacturers. The attack, requiring a minimum of four rolling codes (traces), guesses a 16-bit window of the keystate and calculates the correlation score between the observed and calculated keystate. Next, they take the windows with the highest correlation score and repeat the previous step with the window covering an extra bit until they have recovered the unknown 32 bit of the key. In [17], Benadjila et al. propose a black-box analysis of RKE-specific implementations of Hitag2, referred to as hardened Hitag2, which are not susceptible to the attack described in [70]. Discrepancies in how the ECU forms the Initialisation Vector (IV) lead to little IV variation in consecutive packets, which in turn risks dismissing the correct candidate window early on. They suggest an exhaustive-search based technique requiring only two authentication traces, which recovers the key within 18 hours on a GPU. Verstegen et al. improve on this exhaustive-search based approach in [217]. They propose a highly optimised guess-and-determine attack, which is over 500 times faster and recovers the 48-bit key within approximately one minute.

**VAG RKE** Furthermore, Garcia et al. also reveal the lack of key diversification in several iterations of the VAG RKE system [70]. A first scheme, used in cars until approximately 2005 relies solely on security through obscurity and thus does not incorporate any cryptographic primitives. The second and third scheme introduce a rolling code and rely on a 12 round implementation of AUT64. However, both schemes use a fixed, global master key in all transponders, completely nullifying their purpose. The fourth scheme builds upon the XTEA [140] cipher, however exhibits the same weakness as the earlier schemes, namely the use of a single global master key.

**PKES** The immobiliser, PKE, and RKE systems are often intertwined in modern vehicles. When combined, they are referred to as a PKES system, which allows the vehicle to unlock and start automatically when the legitimate user is near-by the vehicle. Francillon et al. showed that several models with PKES systems are vulnerable to relay attacks in [65]. The emitter, in close proximity to the car, picks up and transforms the short range LF signal into a HF signal. It then sends the transformed signal to the receiver, which performs the opposite operation and relays the amplified LF signal to the car key. The wireless attacks are successful up to a range of 30 m. In [95], Joo et al. address this issue and propose a keyfob radio frequency fingerprinting method to mitigate against relay attacks on keyless entry systems. By analysing the Ultra-High Frequency (UHF) band emitted by a key fob, the authors detect any relay attacks without any false negatives. Their approach is backwards compatible simply by adding a UHF-enabled analysis device to the original system.

## 3.3 CRP bypass techniques

Chip manufacturers typically incorporate a CRP mechanism on their products in order to prevent unauthorised access to the chip's memories. Adopters of the chip along the supply chain (e.g., Tier-X suppliers) benefit from this mechanism in various ways: (*i*) The CRP safeguards the Intellectual Property (IP) contained within the chip. (*ii*) Disabling access to the memory prevents tampering with the flashed program and thus adds to the integrity of the system. (*iii*) In case of chip failure, many (properly configured) CRP mechanisms still allow debug access after providing a specific secret set at flashing time. However, the aforementioned properties do not hold if an attacker can bypass this protection through either software or hardware vulnerabilities. Thus, we give an overview of the literature on fault injection techniques and present the state-of-the-art in CRP bypass attacks.

### 3.3.1 Fault-Injection Techniques

Hardware-based fault attacks induce a fault in on-chip computations, such as skipping an instruction, by changing the physical operating environment of the chip, e.g., the supply voltage. They do not rely on the presence of a software vulnerability. Depending on how much the attack method interferes with the original chip, Skorobogatov classifies fault attacks into the following three categories [187]:

**invasive attacks:** Techniques which physically interact with the chip circuitry, thereby destroying the die in the process. This includes microprobing, which accesses the IC directly, or reverse engineering of the silicon. This type of attack requires the least initial knowledge of the device under attack and often works for a wide range of chips, but is typically only feasible with expensive equipment.

**semi-invasive attacks:** The attack requires access to the chip's die, however the chip remains functional during the process and thus is not destroyed after the attack. Optical fault injection belongs to this category, since it requires direct access to the functioning chip's memories. Since semi-invasive attacks do not require direct physical access to the chip's internals, they are typically useful to explore the device functionality.

**non-invasive attacks:** The attacked chip remains intact during the whole course of the attack. For instance, both voltage or clock glitching operate on the original chip and only require connecting to an external pin. Non-invasive attacks often require intricate knowledge of both the hardware and software under attack, but are achievable with cheap, off-the-shelf hardware.

Ever since the first published fault attack by Boneh et al. on the RSA cryptosystem [21], fault injection has been an active area of research. The literature covers a wide spectrum of hardware-based fault injection methods: the most widely-used techniques include voltage, optical, clock, temperature and electromagnetic fault injection, which we will cover here in more detail.

**Optical F-I** Optical fault attacks illuminate an on-chip transistor which makes it conduct and thus induces a transient fault. Anderson et al. point out an attacker can clear the security bit of a microcontroller by focusing UV light on the security lock cell [6]. Similarly, Skorobogatov et al. use an off-the-shelf laser pointer to flip single bits in the RAM of a PIC16F84 µC [186]. In [172], Samyde et al. use a less powerful laser to read out an SRAM cell in a µC. The photons emitted by the laser carry energy larger than the silicon band gap and thus produce a photocurrent in the p-n junctions, which in turn decreases its resistance. However, the decrease in resistance is only noticeable for closed and not for open channels, revealing the memory cell's state. Next, Skorobogatov introduces *fault masking attacks* on non-volatile memory such as flash and EEPROM in [183]. Instead of targeting the individual memory cells or the data bus on which memory is fetched (which would require $n$ lasers for a n-bit data bus), he focuses the laser on chip area containing the memory control logic. With this semi-invasive attack he disables the memory write and erase operations. In [182], Skorobogatov introduces flash memory bumping; a technique which uses optical fault-injection to force the data bus into a known state. µCs with a verify-only approach compare the data in memory with uploaded data, and report an error if they do not match. By selectively *bumping* certain bits on the data line into a known state, an attacker can brute force the remaining bits of a word. Modern chips integrate countermeasures into their design, such as incorporating physical barriers on the die, which shield the sensitive parts, or including optical sensors to detect sudden photon spikes. In [212], van Woudenberg et al. addresses this and systematically list requirements for successful optical fault injection in modern secure microprocessors (e.g., smartcards), which require accurate equipment. Firstly, they introduce a *pattern based trigger*, which fires on detection of a predetermined signal, the power consumption in this case. Next, they use diode lasers, which are capable to perform multiple glitch attacks, to project optical pulses to the die. Finally, the authors address the *glitch parameter optimisation*, which remains an open research question. For optical fault attacks, the parameters include the location on the chip, the time delay after the trigger and the duration and intensity of the pulse. Since the chip surface must be accessible for optical fault attacks to succeed, they require extensive preparation such as decapsulation and thus are classified as *semi-invasive* attacks.

**Temperature F-I** Tampering with the ambient temperature of a chip can lead to a faulty execution. Research has shown that overheating the chip or cooling it to extreme temperatures are both successful fault injection techniques. On the one hand, as described by Anderson et al. [6] and later shown by Skorobogatov et al. [181], by freezing the chip to a low temperature (e.g., -50°C), volatile data can be retained much longer than what would be deemed secure. This is called the *data remanence effect*, which states that the contents of volatile memory fade away gradually over time instead of disappearing immediately after power down. Skorobogatov points out several security issues the data remanence effect poses on various types of memory devices [185]. An attacker can still recover information from a memory cell even after 100 erase cycles due to residual charge. To illustrate, Samyde et al. recover several Data Encryption Standard (DES) keys from RAM after cooling the chip with a freezing spray [172]. Halderman et al. introduce cold boot attacks in [84] to recover encryption keys from volatile memory and thus bypass the disk encryption. Even up until 10 minutes after power down, the DRAM retained more than 99% of bits when cooled down to -50°C. In [81], Gruhn et al. assess the practicality of these cold boot attacks. They empirically asses the remanence effect on several DDR2 and DDR3 modules. Furthermore, the authors introduce RAM transplantation, where the RAM module is cooled down and then moved to another computer. This defeats the software countermeasures put in place, such as resetting the RAM on boot, wiping the memory when temperature sensors report a temperature below a certain threshold or locking the boot process. Then, in [133], Muller et al. introduce *FROST*, a recovery tool that leverages cold boot to bypass the disk encryption on Android phones. On the other hand, high temperature attacks can induce memory faults on the chip by raising the ambient temperature, as shown in [161]. Govindavajhala et al. exploit this peculiarity to take over a Java Virtual Machine through a soft temperature-induced memory error in [78]. The exposed memory chip, heated by a 50W spotlight bulb, starts showing errors from 100°C, resulting in an attack success rate of roughly 70%. Next, Hutter et al. exploit heating faults to retrieve the private key of an RSA implementation on an AVR chip [90]. Moreover, through extensive heating they burn the (constant) private key into memory, which they can recover even after years. The authors equally characterise the *temperature side-channel*, showing a linear correlation between the emanated heat and circuit activity.

**Electromagnetic F-I** Electromagnetic Fault Injection (EMFI) is a semi-invasive technique which introduces anomalies in a chip's behaviour through electromagnetic signals. Quisquater et al. characterise the electromagnetic side channel in [162]. They show that a chip's electromagnetic radiation holds at least similar information to the power consumption, but can be more precisely directed at certain processor areas. Then, in [161], Quisquater et al. use a camera to inject an eddy current into a microprobe needle, which they use to create a map of the chip's layout. Samyde et al. continue this line of research in [172]. By creating small pertubations on memory cells using EM radiation, they flip several bits in the SRAM and read out several bytes from memory. However, even small pertubations cause whole rows of memory cells to change state, instead of single bytes. Moro et al. propose a study on the precise effects of EM faults on a state-of-the-art microcontroller. They attribute faults on the assembly level to a register level transfer model, and confirm injected faults can alter an instruction's opcode, resulting in a different operation. More recently, Cui et al. direct electromagnetic pulses at modern computers to defeat the TrustZone component on a modern phone [49]. By exploiting so-called *second order* effects, which focus on the interdependence of components on an IC, they significantly reduce the temporal and spatial resolution of EM fault attacks. Furthermore, they present BADFET, an electromagnetic fault injection platform.

**Clock F-I** Clock fault injection techniques introduce one or several shorter clock pulses, or *glitches*, on the external clock line of a μC with the aim to cause a faulty instruction. Two parameters are important to set up a successful clock fault injection:

the precise timing and duration of the anomaly. Kommerling et al. classify clock glitching as a the simplest and most practical glitch attack at the time in [109]. By temporarily increasing the clock frequency around a targeted instruction, certain flip flops sample their input before the propagation delay, causing them to latch faulty data. They note that chip manufacturers can mitigate against these attacks by introducing randomness at the clock cycle level. Agoyan et al. address the issue of how to efficiently generate a clock glitch in [5]. By generating two delayed clocks using the Delay Locked Loop (DLL) on a Xilinx FPGA, they can temporarily increase the clock frequency. On trigger, they introduce the clock glitch by combining the rising edge of the first delayed clock with the falling edge of the second. The authors successfully generate one and two bit errors on an AES implementation, leading to full secret key recovery with a 90% success rate using Giraud's DFA technique [74]. Then, in [12], Balasch et al. characterise the exact effect of clock glitches on an 8-bit AVR μC. The authors show that by introducing a clock glitch, they can affect the loading of the next instruction, causing some bits to change and thus resulting in a different opcode. Interestingly, multi-cycle instructions on architectures with a multi-stage pipeline are less complex glitch targets, making it easier not to disturb the data flow since they fill up the whole pipeline. Finally, the authors note that clock faults behave deterministic and thus are able to reproduce their effects.

**Voltage F-I** Voltage fault injection is a non-invasive glitch technique which temporarily introduces an anomaly in the voltage supply of the target chip. It does not require expensive lab equipment: open-source projects such as the Chipwhisperer [152] and the GIAnT [154] significantly lower the entry barrier for voltage glitching. Kommerling et al. highlight the practical benefits of power fluctuation as a fault injection technique in [109]. They note that due to the lack of large on-chip capacitors and the inability for voltage treshold sensors to detect fast transients, smartcards make a good target for voltage fault injection. In the following decades, the literature covers a plethora of voltage glitch attacks on smartcards, microcontrollers and cryptographic primitives, e.g., [13, 14, 101]. A novel glitching technique using an N-channel MOSFET, *crowbar injection*, even achieves better temporal accuracy than clock glitching on an AVR chip [150]. Researchers have triggered voltage faults even through software, breaking secure properties of Intel SGX [135] and AES modules on an FPGA [112]. To reduce the time for determining glitch parameters, several parameter optimisation strategies for voltage glitching have been proposed in the literature. Carpi et al. investigate several parameter search strategies on smartcards, of which a generic zoom-and-bound approach proves the most effective, although a genetic algorithm also generates promising results [32]. Picek et al. follow up on this in [158] and propose an improved solution based on a genetic algorithm. They propose a novel fitness evaluation and improve on the generic crossover function. Finally, in [24], Bozzato et al. continue along the same line and focus on the glitch shape in their genetic search strategy. Note that all these strategies treat the chip under attack as a black box and do not take the executed firmware binary into account.

### 3.3.2 Hardware-based CRP bypass

Researchers have used the aforementioned fault techniques to bypass CRP mechanisms on various µCs, starting with the work presented in [184], which breaks the copy protection of various µCs using both clock and power glitching. In 2002, Skorobogatov et al. showed that they could change a single bit in an SRAM array on a PIC16F84 microcontroller using an off-the-shelf laser pointer [186] and use this effect to bypass CRP. In [182], Skorobogatov introduces flash memory bumping; a technique which uses optical fault-injection to force the data bus into a known state. Skorobogatov used this to extract read-protected memory from a NEC 78K0/S microcontroller and a Actel ProASIC3 FPGA. Similarly, researchers have targeted the fuses containing the readout protection bits with UV-C light [27, 149, 195]. In [149] Obermaier et al. bypassed the CRP of the STM32F0 by injecting a fault to flip one bit in the 16-bit value encoding the CRP level, which ranges from 0 to 2 (0 indicating protection disabled). Because CRP level 2 and 1 only differ by one bit, they downgraded the security level to CRP level 1, which enables the

SWD interface. The researchers then discovered two additional vulnerabilities in SWD First, they introduced cold boot stepping, a technique which reconstructs the program control flow based on SRAM snapshots: even though the flash memory is read-protected in this scenario, the SRAM is not, enabling an attacker to take a snapshot for every specific amount of cycles. Second, they found a timing-based race condition vulnerability in the read requests, which in combination with cold boot stepping lead to a full attack on CRP 1. Obermaier et al. analyse the security of a real and several counterfeit STM32 chips, uncovering various software and hardware vulnerabilities in the debug interface and chip design [148]. In [151], O'Flynn et al. analyse the readout protection mechanism on SPC56xx chips, used in several recent ECUs. If a high level is input on the boot mode pin on reset, the bootloader performs a password check and if successful, an external device can download additional code to memory. The authors bypass all three levels of readout protection (based on the password) on the MPC56xx series through EMFI. A shunt resistor embedded in a development chip reveals when the bootloader performs the password check, upon which the fault is injected.

Other research has shown that an attacker can disable CRP on chips through voltage fault injection [24, 116, 71, 33, 34]. By changing the voltage level (*i.e.*, glitching) at the time the chip evaluates or loads the CRP value, they can bypass the protection mechanism and gain read/write access to the flash memory. For instance, the CRP value is initially loaded from flash into the RAM in NXP LPC μCs, and the respective checks can be manipulated through glitches [71]. Instead of glitching the readout protection enforced by the bootloader, Milburn et al. direct their efforts towards the diagnostic layer. They read out 0x40 bytes at a time by glitching the `readMemoryByAddress` UDS service, resulting in a total firmware readout time of 3 days. In [24], Bozzato et al. propose a genetic algorithm to optimise the glitch parameters and waveform by coupling a Direct Digital Synthesis (DDS) device with an amplifier in order to inject arbitrary waveforms. They show and evaluate the effectiveness of their technique by bypassing the copy protection on STM32, MSP430 and 78K0(/R) chips. Another class of attacks arises where a voltage glitch induces a software vulnerability such as a buffer overflow as described in [118]. Roth et al. show the practical challenges of voltage glitching attacks: extracting the private key from an STM32-based hardware cryptocurrency wallet required a 3-month profiling phase to determine the correct glitch parameters (offset, width and voltage, cf. Section 2.3.3) through exhaustive search [169]. In [105], Khan covers several implementation flaws in chip readout protection. In particular, many developers disable or reconfigure chip debug interfaces (e.g., JTAG) at runtime. However, the author points out that an attacker with physical access can latch on to these interfaces at the early boot stages. Equally, the author notes that bootloaders could leak sensitive information, such as passwords, through various side channels (e.g., a timing side channel). Logic errors, such as incorrect address blacklists for enforcing the read protection, can lead to (partial) compromise of the firmware. Finally, the author proposes several good practices to implement secure bootloaders: (*i*) Any embedded bootloader should receive independent scrutiny before going into production. (*ii*) Use of secure boot, which ideally cryptographically authenticates any software running on the device. (*iii*) Minimising the bootloader attack surface.

To summarise, Yuce et al. give an overview of commonly used types of hardware-controlled fault injection attacks in [225].

### 3.3.3 Software-based CRP bypass

Bootloaders providing reprogramming functionality at the very least require code to handle communication and to read/write flash memory. This has lead a number of published software vulnerabilities. Temkin et al. found a stack overflow in the USB code of NVIDIA's Tegra bootloader, leading to code execution and CRP bypass [197]. Goodspeed et al. make use of the fact that the bootloader is placed at a fixed memory location when blindly exploiting stack-based buffer overflows in embedded application code [77].

Timing dependencies in the code that verifies a password protecting the access to the bootloader (anti-pattern A5) have lead to the compromise of the M16C [160] and MSP430 [75] μCs. Additionally, through single-stepping instructions and reading RAM or register contents over a debug interface, researchers have bypassed the copy protection of NRF51822 [26], STM32F0 [149] and STM32F1 [173] chips (anti-pattern A2).

Even when the debug interface is properly protected, it has been shown that attackers can recover sensitive data from RAM or data flash once the program flash is erased and thus the readout protection reset [115, 76] (anti-pattern A4). Similarly, if the chip allows erasing per flash sector, an attacker can overwrite the boot section with a program that reads out the firmware [122] (cf. Section 5.4.3 and anti-pattern A3).

## 3.4 Embedded Device Analysis

Over the past decade, two main objectives have driven security research into embedded device firmware: vulnerability discovery and reverse engineering. The literature commonly divides (embedded) program analysis techniques into two categories: *dynamic* and *static* analysis. Whereas the dynamic approach executes at least part of the code, a static approach solely analyses the program binary. Whereas many of the subsequent techniques are well established on systems with commodity OSs (*i.e.*, Linux or Windows), their application to embedded firmware often lags behind. We refer the reader to [130] for a more thorough overview of the specific challenges embedded firmware analysis poses.

### 3.4.1 Dynamic Analyis Techniques

**Firmware Rehosting** Avatar [131, 226] and Surrogates [111] introduced the concept of hardware in the loop analysis. This methodology allows dynamic analysis to be performed in a way that handles many of the complexities of the underlying embedded devices, such as interrupts and peripherals. It enables hybrid analyses where a fast host can emulate the majority of the firmware and defer to the device only for I/O, interrupts and interaction with peripherals. PANDA [55] addresses this problem by shifting firmware execution to an emulator and records fine-grained execution traces that can be replayed whilst performing different analysis passes. In order to remove the need for device interaction, many approaches attempt to completely *rehost* the firmware using a generic emulator and provide peripheral models and simulated interrupts to simulate the hardware. To a greater [60, 121] and lesser extent [83, 85] these models can be automatically generated. P2IM bridges the gap between fuzzing and firmware emulation by automatically generating *approximate* µC *emulators* based on their firmware. Their technique relies on the fact that emulation succeeds as long as peripherals return adequate inputs when needed, e.g., for passing a system register check. They achieve this by identifying several types of peripheral registers and modeling their access patterns. Mera et al. present DICE, a drop-in component on top of existing dynamic analysis frameworks (e.g., P2IM), to address the difficulty of emulating firmware using Direct Memory Access (DMA). They identify DMA channels by writes to their respective control registers, and provide fuzzing input to the DMA input channels (e.g., memory buffers used by the firmware) when the firmware reads from them. In [83], Gustafson et al. leverage recordings from the original hardware to model peripherals and interrupts. Based on these they cluster peripherals by memory regions they access and link interrupts to a specific peripheral group. Then, based on the access patterns the authors provide memory models for each peripheral memory location. In [85], Harrison et al. follow a similar approach to emulate and fuzz certain components of Trustzone Operating Systems, which rely on ARM's trusted execution environment. They analyse the coupling of the several hardware and software components, and decide whether to emulate the original component or to insert a stub which mimics its behaviour. Moreover, HALucinator [40] exploits the observation that many devices share the same Hardware Abstraction Layer (HAL) to access peripherals and low-level functionality. By hooking this layer and emulating peripherals, their technique supports many devices with the same HAL, which is often the case for devices in the same family. Milburn et al. point out several challenges to ECU emulation in [124], the main one being the lack of emulator support. Hence, they build a custom emulator for an instrument cluster µC and either stub or model the necessary peripherals. They discover the CANIDs the ECU listens and responds on by taint tracking several CAN registers. Finally, the authors run existing CAN fuzzers on the emulated firmware over a virtual CAN channel.

**Symbolic Execution** Symbolic execution is a widely used technique in software testing and program analysis [106]. This technique symbolises the program input variables and tracks their constraints. Then, it can solve these constraints to generate a viable (concrete) input value that will cause the execution of a new path along the program tree. Several approaches rely on the availability of (part of) the source code to apply this technique to embedded firmware. Davidson et al. [50], for example, show its effectiveness on very small firmware to achieve complete coverage given the source code of the program. They adapt the KLEE [29] symbolic execution engine to work with embedded firmware, of the MSP430 in particular. The authors enhance *state pruning*, which detects when a program's state has been previously analysed, and present *memory smudging*, which recognises and replaces loop counters with unconstrained variables, to enhance the symbolic execution. Other approaches [44, 45] focus on how to handle the nuances of complex firmware and devices with multiple peripherals under symbolic execution. They leverage the fact that source code is sometimes available for portions of firmware, which contains significantly more information about high-level constructs and data-types and can therefore make symbolic execution more tractable. Concretely, Inception [44] merges the LLVM bitcode of high-level source code with that of low-level assembly and library code. This approach preserves the semantics of the source code, while still allowing for complete execution of the firmware (which may include hand-written assembly). Building on Inception, Cortegianni et al. present HardSnap [45], an approach which takes into account the hardware state (*i.e.*, peripherals) and firmware of an embedded device for symbolic execution. It is aimed at firmware developers and thus relies on the availability of hardware and software source code. By creating a full system snapshot, fuzzing and symbolic execution based approaches can rerun the program from a set point. To achieve this, the authors instrument peripherals with an introspection mechanism.

**Fuzzing and Partial Emulation** Other approaches use fuzz testing to assess devices for vulnerabilities. In this case, a *fuzzer* repeatedly mutates input from an initial *seed* and sends it to the device aiming to cause a crash or anomalous execution. A naive approach to this has significant drawbacks as highlighted by Muench et al. [132]. Fuzzing embedded targets requires a device restart after each fuzzing input in order to have a clean state for the next input. Furthermore, a memory corruption does not immediately manifest in a faulty execution, therefore further complicating the analysis. Therefore, most approaches first manually reverse engineer the firmware and then emulate select parts [119, 171, 189, 227]. BaseSAFE [119] builds on the Unicorn engine, a fork of QEMU, and AFL to partially emulate and fuzz baseband firmware. They reverse engineer the baseband RTOS, Nucleus, and pinpoint several handler functions of two Long Term Evolution (LTE) layers to emulate. Similarly, Frankenstein [171] builds on QEMU to emulate and fuzz large parts of the Broadcom and Cypress Bluetooth stacks. FirmFuzz [189] and FirmAFL [227] both focus on emulating Linux-based firmware images. The former leverages static analysis to generate inputs that trigger deep vulnerabilities in the firmware. The latter combines system mode emulation with process emulation to obtain a high fuzzing throughput. Aside from fuzzing and symbolic execution, Chen et al. [37] and Costin et al. [47], partially emulate more high-level (*i.e.*, Linux-based) device firmware and demonstrate even off-the-shelf tools are effective in finding vulnerabilities in some firmware.

**Tracing** Aside from academic work, both SystemView [175] and Tracealyzer [156] support developers in debugging their firmware through trace analysis. However, they require access to the source code for the firmware under test, such that it can be recompiled with software-based instrumentation. As such, the use of these tools for reverse-engineering or analysis of end-user devices is extremely limited.

### 3.4.2 Static Analysis

While the majority of focus has been on dynamic techniques, a small number of approaches have used static analysis. However almost all of these have not been applied to bare metal or monolithic firmware. Costin et al. [46] perform a large scale assessment of downloadable device firmware (mostly Linux-based) and apply simple static analyses to discover a number of vulnerabilities. Shoshitaishvili et al. [177] demonstrate how symbolic execution with human guidance can be used to discover authentication bypass vulnerabilities in a variety of firmware, including binary blob firmware. Redini et al. [164] apply static taint propagation and symbolic execution to discover vulnerabilities in IoT and router firmware across process boundaries. Cojocar et al. [41] use static analysis and machine learning to automatically locate parsing routines in device firmware. Similarly, Thomas et al. [204, 205] use a static approach to locate undocumented functionality in Linux-based firmware.

**Control-flow recovery and disassembly** A significant body of research has contributed to the state-of-the-art in binary analysis and in particular control-flow recovery and disassembly. We refer the reader to the work of, e.g., Shoshitaishvili et al. [179] for a more complete exposition. IDA Pro [88] and Ghidra [209] provide solutions to aid manual reverse engineering by performing function start identification, disassembly, control-flow recovery, and cross-referencing. However, for complex firmware the quality of their analyses is highly dependent on human intervention. Andriesse et al. [7] provide an in-depth analysis of the problems performing disassembly on real-world x86/x64 binaries. They attempt to remedy the situation with a compiler agnostic approach to control-flow recovery [8]. Muhui et al. [94] perform a similar analysis for ARM-based binaries. The authors of Polypyus [66] propose an approach leveraging past reverse-engineering efforts on firmware from the same vendor to aid in control-flow and function start recovery, their tool provides a step forward in addressing a problem which dramatically affects the correctness of disassembly for VLE architectures such as Thumb2. De Goër et al. [51] show the difficulty in discerning between calls and jumps for intra-procedural calls. To achieve this, they dynamically instrument the devices. We note that the complexity for reconstructing the control-flow of an interrupt driven firmware is highlighted perfectly by Tan et al. [196] in their demonstration of a bugdoor hidden in the interrupt handling logic of a firmware, triggered by a specific sequence of interrupts.

**Human-in-the-loop** Tool-centric semi-automated techniques are necessary for handling the complexity of many reverse-engineering tasks, especially when there is hardware-software interaction. This is the case even for *mechanical* tasks such as trace capture. They shine however, when they can provide the creativity aspect that is almost always required when reverse-engineering a hard target. Little published work acknowledges this, yet its effectiveness is undeniable (see, e.g., IJON [11]). To this end, Shoshitaishvili et al. [180] present a "human assisted" cyber-reasoning system, and propose a paradigm shift towards tool-centred, human-assisted approaches.

# Part II

# ECU Firmware Extraction and Analysis

# CHAPTER 4

# BREAKING DIAGNOSTICS: FIRMWARE EXTRACTION OVER CAN

## Contents

This chapter proposes a technique to extract firmware over the automotive network. Using both documented and hidden diagnostic functionality, we bypass the diagnostic authentication mechanism to download code to the ECU and execute it. It is based on the following publication:

*Van den Herrewegen J., Garcia F.D. (2018) Beneath the Bonnet: A Breakdown of Diagnostic Security. In: Lopez J., Zhou J., Soriano M. (eds) Computer Security. ESORICS 2018. Lecture Notes in Computer Science, vol 11098. Springer, Cham.*

## 4.1 Motivation

The functionality of a modern road vehicle is determined by a few dozen ECUs. These are interconnected via one or several CAN buses. Powerful diagnostic protocols are put in place by the manufacturer to update or patch the vehicle in case of malfunction. The most prevalent diagnostic standards are UDS (cf. Section 2.1.3 and [2]) and its predecessor, KWP (cf. Section 2.1.3 and [3]), which provide manufacturers and service technicians with advanced diagnostic features such as upload and download functionality. The main diagnostic access control mechanism is the so-called *seed-key* protocol, a challenge-response protocol used to authenticate diagnostic devices. Even more sophisticated diagnostic protocols such as XCP(cf. Section 2.1.3 and [223]) enable service technicians to fully fine-tune ECUs. The functionality provided by XCP goes beyond that of traditional diagnostic protocols found in ECUs, which a knowledgeable attacker could abuse to take control of an ECU over CAN.

In many cars, diagnostic communication occurs on the CAN bus available on the OBD-II port, which every vehicle commissioned in the European Union since 2004 [54] must be equipped with. However, the automotive network was never designed with an adversary in mind: the CAN bus is an unencrypted and unauthenticated network. Thus, ECUs cannot distinguish diagnostic messages originating from a diagnostic client from messages sent by an adversary. As shown in Chapter 3, previous research has indicated that individual ECUs connected to the internal network of a modern car can be compromised [125, 110]. This becomes even more worrying when combined with a remote exploit, as demonstrated in e.g., [210]. With advanced features such as in-vehicle connectivity becoming the norm in modern cars, the automotive industry needs to shift towards better diagnostic security in ECUs.

## 4.2 Contributions

The contribution of this chapter is three-fold:

- Through reverse-engineering the ECU firmware of three different manufacturers, we recovered the ciphers used in the diagnostic authentication protocol, which we present here in full detail.
- We propose a practical cryptanalysis of each of these ciphers, showing that the diagnostic authentication protocols can be easily bypassed with neglibible computational complexity.
- We propose a generic method to remotely execute code on an ECU by exploiting UDS and XCP features, giving us read/write access to the internal memory of the ECU and its peripherals.

## 4.3 Cryptanalysis of diagnostic protocols

In this section we analyse the ciphers used in the diagnostic challenge-response protocol, which we extracted from ECUs of three different automotive manufacturers. We recovered and analysed the firmware of 13 ECUs in total, comprising 8 different car models. We focused our efforts on modules with a security critical function, such as the Instrument Cluster and Body Control Module (which handle immobiliser functionality and store its secret keys), a Gateway (which separates the critical high speed CAN bus from other low speed buses), and a Telematics Unit (which provides connectivity to the outside world). Next, we revisit the cipher first described by Valasek and Miller in [125] and present new vulnerabilities, making it easy to circumvent in practise. Using the IDA Pro disassembler we have recovered challenge-response ciphers from the firmware of Ford, Volvo, Fiat and Audi ECUs. We present these ciphers and analyse their security.

### 4.3.1 Obtaining and analysing ECU firmware images

On all ECUs we have studied in this chapter, the firmware is located in the internal flash memory of the microcontroller. We lift the firmware from the hardware of these embedded devices through a debug interface, such as JTAG or Background Debug Mode (BDM), which is often exposed on a group of test points on the PCB (cf. Section 2.3.2). Next, we load the firmware into the IDA Pro disassembler on the correct memory address, which is specified in the datasheet of the microcontroller. For microcontrollers that incorporate a paging mechanism, such as the MC9S12XE (used on certain Ford Instrument Clusters and Body Control Modules), we first separate the firmware into chunks equal to the page size of the microcontroller. Once loaded, we can locate the cipher used in the diagnostic authentication protocol by searching for functions that contain constants typical for UDS, more specifically frequently used diagnostic error codes and/or service identifiers. From studying the ECUs in different cars and models, we have noticed the manufacturers and tier-1 suppliers often reuse ECUs running the same or at least a very similar firmware version across different cars and models. Thus, we only need to go through this process once for every ECU type.

### 4.3.2 Analysis of the Ford challenge-response cipher

In this section we perform a cryptanalysis of the Ford cipher, which we have located in the firmware of several Ford ECUs but also in some Volvo units, depicted in Table 4.1, through our reverse engineering efforts. We introduce the cipher and demonstrate how an attacker can break it by means of an attack over CAN. According to [61], roughly 10% of cars sold in the United Kingdom are Ford, accounting for ~200000 cars per year.

**Cipher details.**

Both the challenge `C` and response `R` are three bytes in the authentication protocol on Ford ECUs. The cipher uses a slightly modified version of the Galois LFSR with an internal state of 24 bits, which is initialised with a constant (`C541A9`) stored in the firmware (flash memory) of the ECU. The output bit of the LFSR is XORed with a bit from a 64-bit input register `I` consisting of a 40-bit *secret* `S` and the 24-bit challenge `C`. Figure 4.1 depicts

Table 4.1: ECUs on which we examined and identified the Ford cipher.

| **Make** | **Year** | **Model** | **ECU** |
|---|---|---|---|
| Ford | 2010 | Focus MK2 | Body Control Module<br>Instrument Cluster |
| | 2012, 2014, 2016 | Focus MK3 | Body Control Module<br>Instrument Cluster |
| | 2008 | Fiesta MK6 | Instrument Cluster |
| | 2013, 2014, 2015, 2017 | Fiesta MK7 | Instrument Cluster<br>Body Control Module |
| Volvo | 2015 | V50 | Telematics Unit |

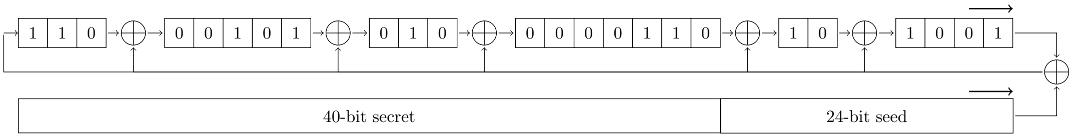


Figure 4.1: Initial state and structure of the Ford LFSR used in the challenge response authentication.

the structure of the modified Galois LFSR, while Definition 1 details the input bit of the cipher in round $i$. The cipher runs for 64 rounds: in the first 24 rounds, the challenge is shifted into the internal state, after which the cipher absorbs the 40-bit secret into its internal state. In each round, the XOR of the output bit of the LFSR and the input bit of the register is fed back into the tapped bits. The final response is derived from the 24-bit LFSR-state by permuting the nibbles of the state, as shown in Definition 2.

**Definition 1.** Given challenge $C$ and secret $S$, input bit $R_i$ in round $i$ is defined as follows.

$$R_i = \begin{cases} \mathrm{C}_{i}, & \text{if } i < 24 \\ \mathrm{S}_{24-i}, & \text{if } 24 \leq i < 64 \end{cases}$$

**Definition 2.** Let the nibble representation of the internal state $Y$ be $n_5, \ldots, n_0 = Y[2], \ldots, Y[0]$. Then the permutation $P_1(n_5, \ldots, n_0) : \mathbb{F}_2^{24} \rightarrow \mathbb{F}_2^{24}$ is defined as follows.

$$P_1(n_5, \ldots, n_0) = \begin{pmatrix} n_5 & n_4 & n_3 & n_2 & n_1 & n_0 \\ n_2 & n_1 & n_3 & n_5 & n_0 & n_4 \end{pmatrix}$$

### Weaknesses.

The internal state of the LFSR contains merely 24 bits of entropy. What is even worse, we have observed the same start state and tapping sequence across *all* ECUs we have studied. With no added entropy from a varying start state or tapping sequence, only the 40 bit secret is unknown to an attacker. Through empirical tests we discovered that only the first 24 secret bits shifted into the internal state add entropy. In the subsequent 16 rounds we can set the input bit to zero, making the cipher a standard Galois-LFSR. One valid challenge-response pair enables an attacker to retrieve 24 bits of the secret, and thus recover the structure of the cipher. The attacker can obtain a valid challenge-response pair by making a diagnostic device authenticate to the ECU, which Valasek and Miller demonstrated in [125]. The cipher, however, can be broken even without knowledge of a challenge-response pair.

### Attack over CAN.

We demonstrate how an attacker can recover the secret used in this cipher for a particular ECU without knowledge of any successful authentication pairs. Access to the diagnostic interface of the ECU (e.g., over CAN or through the OBD-II port) is the only prerequisite for this attack.

**Delay mechanism** The UDS standard specifies an error code which indicates a delay timer is active on the ECU in case of too many failed security access attempts. However, the specifics of this mechanism are left up to the manufacturer. Many ECUs implement this rate limiting functionality and disable the security access service temporarily after a certain amount of failed attempts. An attacker can bypass this by requesting a soft reset using the `ECUReset` diagnostic service, which resets all timers and variables. Following a reset the attacker must request a new diagnostic session before they can request a new challenge.

**Recovering diagnostic secrets on Ford and Volvo ECUs** We conducted our attack both on a 2012 Ford Body Control Module (BCM) and a 2015 Volvo Telematics Unit. These particular units do not implement the delay mechanism after a failed security access attempt. Once we request a diagnostic programming session, the units remains in programming mode until no further diagnostic messages are detected for a certain period (~5s). Each security access attempt requires four CAN messages: a challenge request and reply followed by a response and a final message indicating whether the response was valid or not. All CAN frames are 8 bytes for the Ford diagnostic packages, making a physical CAN frame on the bus 135 bits in the worst case, with stuffing bits taken into account [142]. On the BCM, the diagnostic interface is available on the high speed CAN network, which runs at 500 kbit/s. One security access attempt takes four CAN frames or maximum 540 bits, so with a bitrate of 500kbit/s that makes for a minimum of 1.08ms per attempt, calculation time or other delays not taken into account. Since we reduced the complexity from $2^{40}$ of a brute-force attack to only $2^{24}$ attempts, this results in a search time of approximately 5 hours in the best case scenario. Due to all other delays, the attack we implemented took approximately 15 hours. We would like to emphasise that, since all ECUs use the same secret, an attacker only needs to do this once.

### 4.3.3 Analysis of the Fiat challenge-response cipher

Through reverse engineering the firmware of both a current Fiat Body System Interface (BSI) and its predecessor, used in cars before 2012, we have extracted the following cipher used for the security access service. We present the cipher used in the older Fiat BSI for security level 1 and discuss flaws in the design and key generation process.

#### Cipher details

Both the challenge and response are 32 bit in the Fiat implementation of the security access service. The cipher uses two 16 bit LFSRs, both with tapped bits as depicted in

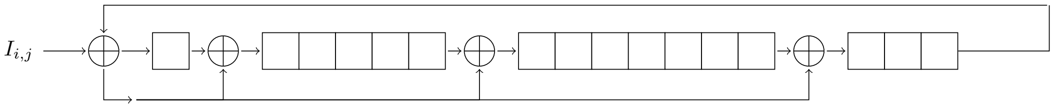


Figure 4.2: Structure of the LFSR used in the diagnostic access control mechanism in Fiat ECUs

Figure 4.2. Both LFSRs absorb one input bit in each round, as detailed in Definition 4. The cipher runs for 24 rounds: in the first 8 rounds different constants ($S[0]$ and $S[2]$) are shifted into each state, whereas in the remaining 16 rounds the cipher absorbs one bit of the preprocessed challenge bytes into the state. Finally, the 32 bit response is derived from the LFSRs by combining the 16-bit internal states.

**Definition 3.** For a given byte $b$, the permutation $P_2(b) : \mathbb{F}_2^8 \to \mathbb{F}_2^8$ is defined as follows.

$$P_2(b) = \begin{pmatrix} b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \\ b_3 & b_0 & b_6 & b_1 & b_7 & b_4 & b_2 & b_5 \end{pmatrix}$$

**Definition 4.** With given challenge $C$ and secret bytes $S[0], \ldots, S[3]$, input bit $I_{i,j}$ in round $i$ for LFSR $j$ is defined as follows.

$$\begin{aligned} I_{i,0} &= (C[3] \oplus S[1], C[1] \ggg 5, S[0])_i \\ I_{i,1} &= (C[0] \oplus S[3], P_2(C[2]), S[2])_i \end{aligned}$$

**Attacking the cipher.**

There are several issues in the design and secret generation of the cipher. The cipher uses two 16-bit LFSRs instead of one 32-bit LFSR, which reduces the entropy added by the tapped bits and start state significantly. An exhaustive search over the secret space would take $2^{48}$ tries, since an attacker must guess the 16-bit start state, the 16-bit tapping sequence and the 8-bit constants $S[0] \ldots S[3]$. However, Table 4.2 depicts the constants found in the firmware of two different Fiat ECUs. Only the tapped bits, $S[0]$ and $S[2]$ differ, while all other bytes are kept constant. The nibbles of $S[0]$ and $S[2]$ are reversed in the firmware of the ECUs. Only the tapped bits in the LFSR are significantly different across the two different ECUs, which reduces the time of an exhaustive search to only $2^{16} = 65536$ attempts.

We have implemented this attack on a Fiat Grande Punto BSI. The diagnostic interface of this unit is available on the high-speed CAN bus, which runs at 500 kbit/s. The ECU enables a delay timer after receiving two unsuccessful security access attempts, which lasts $10s$. However, to circumvent this delay it suffices to establish a new default diagnostic session and immediately thereafter request a new programming session, which resets the timers on the ECU. Thus, every two security access attempts require 12 CAN frames: a programming mode request and response, four frames for obtaining and validating a challenge-response pair (which we do twice) and finally a default mode request and response. This makes for an average of 6 frames per attempt, which comes to a maximum of 810 bits (including stuffing bits) on the CAN bus. For the reduced search space of 65536 attempts this results in a minimum search time of $106s$. The attack we implemented took just over an hour, which is mostly due to the delay incurred when changing from and to a programming session. An attacker only needs to perform this attack once, since diagnostic secrets are shared across similar types of ECUs.

Table 4.2: Secrets found in the firmware of two different Fiat ECUs

| **ECU** | **S[3]** | **S[2]** | **S[1]** | **S[0]** | **taps** | **start state** |
|---|---|---|---|---|---|---|
| Fiat BSI 2012+ | 7A | 34 | DC | 12 | 8408 | FFFF |
| Fiat BSI 2012− | 7A | 43 | DC | 21 | 3423 | FFFF |

### 4.3.4 Analysis of the Volkswagen Group cipher

Through analysing firmware of both Volkswagen and Audi ECUs, we reverse engineered the ciphers used in a 2009 Audi Gateway Control Unit and a 2010 VW Passat Instrument Cluster. The implementation of the cipher in these Volkswagen Group (VAG) ECUs goes as follows. Each ECU contains the same algorithm which interprets a sequence of bytes stored in the firmware as commands on a 32 bit internal state, which is initialised with the randomly generated challenge `C`. Subsequently, the algorithm reads the sequence of bytes, which are parsed as opcodes for the cipher. Each opcode denotes an operation on the internal 32-bit state, with the five basic operations being: rotate the state to the left/right, add/subtract a constant to/from the state and XOR the state with a constant. Based on this information we present the cipher we extracted from the Audi Gateway Control Unit and assess its security.

**Algorithm 1** Audi gateway challenge-response algorithm

```
function CHALLENGE-RESPONSE(C)                          ▷ With C - 32-bit challenge
    $S = C$
    for $i$ in $\{0 \dots 10\}$ do
        $S = S \lll 1$
        if $i \in \{0, 2, 6, 7\}$ then
            if $S_0 == 1$ then                          ▷ For rounds 0, 2, 6 and 7
                $S_0 = 0$                               ▷ Clear the feedback bit
                $S = S \oplus$ 04C11DB7                 ▷ XOR the tapped bits
            end if
        else
            if $S_0 == 1$ then
                $S = S \oplus$ 04C11DB7
            else
                $S_0 = 1$                               ▷ Set the feedback bit
            end if
        end if
    end for
    return $S$
end function
```

**Cipher details.**

Algorithm 1 details the cipher, which runs for 10 rounds. In each round, the cipher rotates the state to the left. The cipher is a standard Galois LFSR: if the feedback bit is set, a constant (the tapped bits, *i.e.*, `04C11DB7` in the code) is XORed into the state. Depending on the round, the feedback bit is either set or cleared.

**Weaknesses.**

Since the internal state of the cipher is equal to the generated challenge, only the 32-bit tapping sequence adds entropy to the cipher. An attacker with access to one challenge-response pair can recover this 32-bit constant by performing an exhaustive search over the 32-bit secret space. It should be noted that the flexible nature of the structure of the cipher makes it more difficult for an attacker to recover the secrets in different ECUs. Indeed, in several VW Instrument Clusters we found that the cipher runs for a different number of rounds and XORs the state with multiple constants, making the cipher more secure.

Additionally, we identified a supplementary security issue in the firmware of this particular unit: if the diagnostic client provides an invalid response, the ECU performs an extra check, which compares the response to a hardcoded value (*i.e.*, `CAFFE012`). The diagnostic tool is authenticated if it provides this value as the response. Regardless of existing vulnerabilities in the cipher, a hardcoded backdoor on the ECU introduces extra security implications.

## 4.4 Remote code execution over CAN

The ciphers we studied in Section 4.3 are in place to protect the ECU from unauthorised access. Once a diagnostic device is authenticated, the ECU unlocks privileged diagnostic functionality, part of which is in place to execute more advanced diagnostic protocols like XCP. Despite its widespread use in the automotive industry, we failed to locate the XCP protocol in the firmware of the ECUs we studied. Instead, we found that the OEM enables a download of the XCP stack to the RAM of an ECU through various diagnostic services. Piggybacking on this required functionality for the XCP protocol, we have identified a generic approach to execute arbitrary code on an ECU over the CAN bus. Through our own reverse engineering efforts we have encountered this mechanism in ECUs made by several manufacturers. Provided that an attacker can bypass the access control mechanism of the diagnostic protocol as shown in Section 4.3, the only prerequisite is that they can send and receive messages on the CAN bus. An attacker with access to the OBD-II port or who has compromised an ECU on the network, such as the Telematics Unit, can abuse this functionality to control or reprogram additional ECUs.

The outline of this section is as follows. After specifying the general method to execute code on an ECU, we show how an adversary with access to the CAN bus can abuse this mechanism to gain read/write access to the firmware of ECUs of several manufacturers. From now on we will refer to the piece of binary code that is sent to the ECU as the *secondary bootloader*.

### Downloading sequence

Figure 4.3 shows the typical sequence of diagnostic messages required to execute the secondary bootloader on an ECU. Firstly, the diagnostic client must request a programming session. Until the client authenticates itself to the ECU, security critical functionality, such as downloading bytes to memory, remains unavailable. Once authenticated, the client can carry out certain checks and assertions about the ECU. These usually include reading out the software version and part number of the module as the secondary bootloader is dependent on the architecture and available peripherals of the microcontroller embedded on the ECU. The client can transfer the secondary bootloader to the ECU through the download services provided by the running diagnostic protocol. Finally, the client requests a routine control either before or after the download (dependent on the manufacturer) in order to redirect the program flow to the secondary bootloader, which

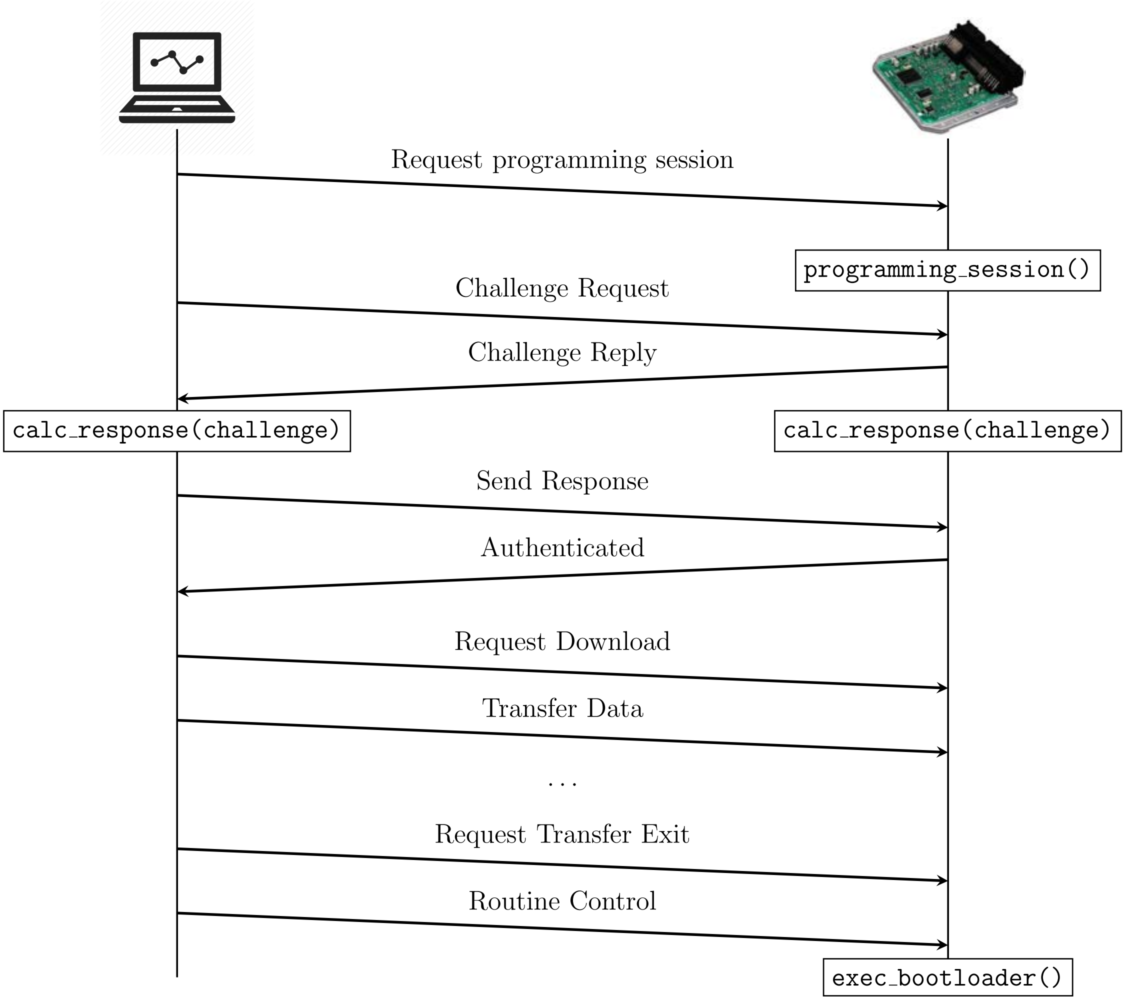


Figure 4.3: Download and execution process of the secondary bootloader from a diagnostic client to an ECU

now resides in RAM.

### Memory limitations.

The ECU only reserves a small area in RAM for the secondary bootloader, which usually suffices if the downloaded code performs a simple task (such as updating a variable in memory). Otherwise, the bootloader can contain additional functionality (e.g., CAN drivers) to download additional code into the RAM of the unit over the CAN bus or other peripherals.

### 4.4.1 Use case: changing the odometer on a Ford Instrument Cluster

To illustrate the capabilities of this secondary bootloader, we have changed the odometer value on a 2016 Ford Focus Instrument Cluster (IC) through the secondary bootloader. The download of the secondary bootloader goes as follows for all Ford and Volvo ECUs we have analysed. With the ECU in a programming session and our device authenticated, we send a `requestDownload` message. The request has two arguments: the download address, which is located in RAM, and the size of the bootloader. If the microcontroller uses a paging mechanism, the address consists either of a page number and address within the page, or a physical address. Subsequently, we can transfer the secondary bootloader using the `transferData` service, after which the ECU expects a `requestTransferExit` message. Finally, to execute the downloaded code, we must send a `routineControl` message. Arguments to this message are the routine identifier, which is `0301`, and the exact address where the microcontroller should jump to. The mileage on this Instrument Cluster is stored on an external Electrically Erasable Programmable Read-Only Memory (EEPROM) chip, namely the M95320 manufactured by ST Microelectronics. The main microcontroller, a Renesas $\mu$PD70F3425, is connected to the EEPROM chip through a Serial Peripheral Interface (SPI). Once we identified the pins used for the serial communication with the EEPROM chip, we managed to arbitrarily reduce the mileage by writing the desired value to the memory locations where the mileage is stored. Multiple ECUs store the mileage in a modern car, meaning that an attacker must repeat this process for all relevant ECUs if he wishes to successfully tamper with the mileage in a car.

It should be noted that Valasek and Miller first documented this bootloader mechanism to reprogram a Ford Smart Junction Box in [125]. There are several differences to the sequence denoted above compared to what Valasek and Miller describe. Firstly, the address the authors specify in the download request to the ECU is zero, which makes the ECU download the code to a predefined address in RAM. Subsequently, the authors call a routine control with identifier `0304`, making the ECU jump to the same predefined address as the download. Finally, the code is only executed if the first four bytes of the secondary bootloader are equal to a value stored in the firmware of the ECU. We have only encountered this 'security' feature in one of the Ford ECUs we analysed.

### 4.4.2 Use case: reprogramming a Fiat Body System Interface

We have analysed the reprogramming process for both a current Delphi Fiat Body System Interface (BSI) and its predecessor, which are deployed in a range of Fiat vehicles. Execution of the secondary bootloader goes as follows for both Fiat BSIs. The ECU must be in a programming session and 'unlocked' for security level 1, following the steps from Section 4.3. In order to execute the downloaded code, we must first write the identifiers with ID's `F184` and `F185` through the `writeDataByIdentifier` service. This sets a flag in memory necessary for the following routine control to complete successfully. Next, we must execute the `eraseMemory` routine control with arguments the identifier (`FF00`), the start address and end address of the memory area in RAM to which we will download the code. In order to make the microcontroller jump to the code, it is crucial that this range is equal to the size of the downloaded data. Otherwise, the download will terminate normally but will not result in a jump to RAM. If all prerequisites described above are met, the microcontroller will jump to a predefined address in RAM after the last `TransferData` request. This address is set in the firmware and is dependent on the memory layout of the microcontroller as the bootloader always resides in RAM. Hence, in order to redirect the program flow to our code, this predefined address must be contained within the download range of the bootloader. Listing 4.1 shows the required diagnostic messages to execute the bootloader.

While the microcontroller runs on a 32-bit architecture, both addresses required as arguments in the routine control preceding the download are only 3 bytes long. The ECU translates these by prepending them with `ff`, resulting in an address located in RAM. Before the ECU executes the downloaded code, it activates the watchdog timer in reset mode, which generates an unmaskable reset interrupt when the timer overflows, making the microcontroller reboot. The secondary bootloader can circumvent this mechanism by resetting the timer before an overflow occurs, implying that the unit will only resume its normal functionality once the bootloader performs a manual reset, for instance by jumping to the reset vector.

Listing 4.1: Executing the secondary bootloader on a Fiat BSI. The client transmits messages on CAN ID `18da40f1`, while the ECU responds on ID `18daf140`.

```
0x18da40f1    2 10 2                   Programming session
0x18daf140    6 50 2 0 32 1 f4 0
0x18da40f1    2 27 1                   Security access
0x18daf140    6 67 1 81 6e e7 f8 0
0x18da40f1    6 27 2 ac eb 3e 3e
0x18daf140    2 67 2 78 0 0 0 0
0x18da40f1    10 10 2e f1 85 1 a8 bc   Write data by ID
0x18daf140    30 0 0 ff ff ff ff ff
0x18da40f1    21 ad cf cf ce ce c9 ca
0x18da40f1    22 13 6 21
0x18daf140    3 6e f1 85 ff ff ff ff
0x18da40f1    10 10 2e f1 84 1 a8 bc   Write data by ID
0x18daf140    30 0 0 ff ff ff ff ff
0x18da40f1    21 ad cf cf ce ce c9 ca
0x18da40f1    22 13 6 21
0x18daf140    3 6e f1 84 ff ff ff ff
0x18da40f1    10 a 31 1 ff 0 ff ca     Routine control
0x18daf140    30 0 0 ff ff ff ff ff
0x18da40f1    21 a0 ff cf 9f
0x18daf140    4 71 1 ff 0 0 0 0
0x18da40f1    10 b 34 0 44 0 ff ca     Request download
0x18daf140    30 0 0 ff ff ff ff ff
0x18da40f1    21 a0 0 0 5 0
0x18daf140    4 74 20 4 2 ff ff ff
0x18da40f1    10 22 36 1 e0 7 60 1     Transfer Data
                       ...
```

## 4.5 Building a firmware modification and extraction framework

We build on this secondary bootloader and the diagnostic primitives enabling it to propose a firmware modification and extraction framework. Using the procedure detailed in Section 4.4, we can execute arbitrary code on any ECU that implements this mechanism. The code downloaded to the ECU is binary machine code, so at the very least we must know the architecture of the ECU. Many microcontrollers used in ECUs are automotive-grade microcontrollers and thus incorporate at least one on-chip CAN interface. This framework aims to transmit the firmware over CAN so the code must contain a minimal microcontroller-specific CAN driver with transmitting capabilities. Table 4.3 lists the ECUs on which we implemented this framework, along with the incorporated microcontroller and the architecture on which it runs. We built a cross toolchain from the GNU GCC source to compile our code for each architecture we encountered.

**Downloading and executing the code.**

The ECU only accepts downloads to a specific area in RAM which varies in different ECUs. Additionally, some units only accept a `RequestDownload` message with a 4 byte address and a 4 byte size, while others are more flexible. UDS provides a set of common negative response codes. If the ECU receives a request with the incorrect format, it replies with a negative response with code `13`, which means *incorrect message length or invalid format.* Contrarily, if the format of the request is correct but the address or size is not within the correct range, the unit responds with error code `31`, indicating *request out of range.* The ECU does not limit the amount of unsuccessful download requests, so we can find this address by covering the complete address space of the microcontroller. Provided that we know the memory layout of the microcontroller, we can limit the range significantly since the address is located in RAM. To further reduce the range, we can increment the address by 0x10 each time while the size remains constant. With a common

Table 4.3: ECUs on which we implemented the firmware extraction framework

| Make | Year | Model | ECU | Microcontroller | Architecture |
|---|---|---|---|---|---|
| Ford | 2012 | Focus MK3 | Body Control Module | MC9S12XEP768 | HCS12X |
| | 2012, 2014, 2016 | Focus MK3 | Instrument Cluster | $\mu$PD70F3425 | V850E |
| | 2008 | Fiesta MK6 | Instrument Cluster | MC9S12HZ256 | HCS12 |
| | 2013, 2014, 2015, 2017 | Fiesta MK7 | Instrument Cluster | MC9S12XEQ384 | HCS12X |
| Volvo | 2015 | V50 | Telematics Unit | SH7267 | SH2A [1] |
| Fiat | >2012 | 500 | Body System Interface | $\mu$PD70F3379 | V850E1 |
| | <2012 | Grande Punto | Body System Interface | $\mu$PD70F3237 | V850E1 |

[1] We failed to extract the firmware from this unit because we did not have access to a CAN driver

ECU RAM size of 128KiB, that makes for a maximum of 8192 attempts.

We can transmit the firmware of an ECU over CAN by dereferencing a pointer and transmitting it until all valid addresses are covered. It suffices to jump to the reset vector to resume normal operation of the ECU. Additionally, we can modify certain crucial parts of the firmware from within the secondary bootloader.

**Gaining access to all diagnostic security levels.**

In order to be able to authenticate to the ECU on all security levels, an attacker must only recover one secret, namely the secret required for downloading the secondary bootloader to the ECU. In the ECUs we have analysed this was always security level 1 in programming mode. The bootloader can extract the firmware, which includes the cipher secrets for additional levels of security. This renders the multiple levels of security defined in diagnostic standards obsolete, provided that an attacker can locate the secrets in the firmware of the ECU.

## 4.6 Mitigation

The only security measure preventing an attacker from downloading code to the unit is the security access service. It is therefore crucial that the challenge-response protocol implemented by the manufacturer is cryptographically sound. Khan [104] proposes the use of the Advanced Encryption Standard for the challenge-response protocol. Given the keys are diversified per car and ECU this would enhance the seed-key security significantly. However, since AES is a symmetric key encryption scheme, the encryption key must be stored in the firmware of the ECU. Unless special hardware is used to protect against reading this encryption key, an attacker can recover the secret key and use it on other ECUs which employ the same key.

A public-key based approach would mitigate the key diversification issues and does not require additional hardware. When a diagnostic client is connected, no time constraints are in place since the car is meant to be stationary during diagnostic maintenance. To mitigate the risk of replay attacks, the challenge is 128 bits long. The diagnostic client generates the response by signing the received challenge with its private key. The ECU verifies the response under the public key, which can be stored in the firmware of the unit. With the computational limitations of ECUs in mind, often running on a 32-bit or even 16-bit architecture, the Elliptic Curve Digital Signature Algorithm (ECDSA) with curve NIST P-256 [159] would be a suitable candidate [82], resulting in a response length of 512 bits.

Moreover, to mitigate the risk of unauthorised code execution on the ECU, the manufacturer can take a similar public-key based approach. If the ECU only accepts downloaded code signed with authorised private keys, no attacker can execute code through this mechanism without knowledge of a valid private key. An attacker with access to the firmware could overwrite the public key with their own public key, which allows them to download code to the unit signed with the attacker's private key. However, we argue that an attacker with the possibility to overwrite the public key can equally overwrite any code in the ECU, making the bootloader mechanism obsolete. Even with access to the firmware, an attacker cannot recover any private keys necessary to execute code on other similar ECUs.

Finally, more secure CAN communication would mitigate the risk of an attacker controlling the complete network from a previously compromised node. Radu et al. proposed LeiA [163], a light-weight authentication protocol for ECUs connected to the CAN bus. In order to transmit on a certain CAN ID, a node must have the authentication key corresponding to that identifier. A node transmits a Message Authentication Code (MAC) along with each message. Receiving nodes can check the validity of the sender simply by computing the same MAC. In this scenario, a node would be secure against attacks from the internal network if no other node has the authentication key for its diagnostic CAN ID.

## 4.7 Discussion

### Security of diagnostic authentication mechanisms

All the ciphers studied in Section 4.3 use some form of proprietary cryptography, with an insufficient challenge and response size of 24 or 32 bits, and an equally small internal state of the cipher. We have shown that if an attacker can obtain a challenge-response pair they can then often recover secret keys of the cipher. No time constraints exist when the ECU is connected to a testbench, as described in [127], making a successful attack over CAN possible.

Efficiently generating and diversifying cryptographic keys for each individual car and ECU remains a difficult issue to solve for manufacturers, as shown in previous research [70, 216]. Valasek and Miller raised the issue of diagnostic key diversification when extracting a set of secrets from a diagnostic device. They (re)used these secrets to authenticate to two ECUs under test. We have encountered similar issues for diagnostic secrets. From our experiments, diagnostic secrets are not diversified for ECUs in each car. An attacker who can recover the secrets for one ECU often has access to other ECUs of the same type or function, since manufacturers reuse these across different models.

### Implications

There are several implications of the insecurity of the bootloader mechanism. Firstly, by dumping the firmware of security sensitive ECUs (such as the Passive Keyless Entry or immobilizer), an attacker can recover cryptographic keys necessary to unlock or start the vehicle. An attentive reader might say that an attacker with access to the internal network does not need to recover cryptographic keys. However, Checkoway et al. present an analysis of remote attack services in [36]. More remote vulnerabilities are covered in the literature, such as in e.g., [170, 210, 62]. These are often generic to the model or even make of the car, implicating that if an attacker gains access to a car through one of these generic remote channels, they could read out cryptographic keys specific to that car.

Additionally, an attacker with access to the CAN bus through the OBD-II port, a compromised ECU or maybe by simply pulling a camera or parking sensor can reprogram or even disable connected ECUs. They can escalate an existing vulnerability to take control over ECUs on the same CAN bus as the compromised node, potentially magnifying the impact of a remote exploit. This would make the notion of an automotive worm possible.

## 4.8 Chapter Summary

In this chapter we exposed several vulnerabilities in diagnostic security. Firstly, we demonstrate how an attacker can bypass the challenge-response security used in diagnostic protocols. All the studied ciphers use some sort of proprietary cryptography, namely an adapted version of the Galois-LFSR. 32- or 24-bit challenges and responses and an equally small internal state further add to the insecurity of the ciphers. We demonstrate this by conducting an attack over CAN to recover diagnostic secrets without requiring any challenge-response pairs. Furthermore, we document the secondary bootloader, a piece of machine code which a CAN node can download to the RAM of a connected ECU through various diagnostic functions. An attacker can abuse this mechanism to recover cryptographic keys, adjust variables in memory or simply disable the ECU. Utilising the functionality implemented for this secondary bootloader, we build a generic firmware modification and extraction framework. To conclude, the challenge-response protocol is the main (and often only) access control mechanism on the ECUs we have studied. The proprietary ciphers used in this protocol are substandard, making it possible for an attacker to bypass these and control all peripherals of the microcontroller through the secondary bootloader, which they can download to RAM. Well deployed public-key cryptographic primitives would mitigate both of these issues.

# CHAPTER 5

# OBTAINING FIRMWARE THROUGH ENHANCED EMBEDDED BOOTLOADER EXPLOITS

## Contents

In Chapter 4, we show how an attacker can piggyback on diagnostic functions to download and execute code to dump the firmware once they are able to bypass the access control mechanism. However, for unknown ECUs on which we do not know the challenge response cipher (e.g., for an unknown ECU of a new manufacturer), or simply other µCs embedded in cars, we cannot yet bypass the security. In this chapter, we propose several techniques to bypass CRP mechanisms in bootloaders, which are almost always present on embedded microcontrollers. It is based on the following publication:

*Van den Herrewegen, J., Oswald, D., Garcia, F. D., & Temeiza, Q. (2021). Fill your Boots: Enhanced Embedded Bootloader Exploits via Fault Injection and Binary Analysis. IACR Transactions on Cryptographic Hardware and Embedded Systems, 56-81.*

## 5.1 Motivation

Embedded microcontrollers are at the foundation of our ever-increasingly digital world, steering innovation through data they collect and process. However, with their many advantages and uses come new security concerns. A single vulnerability in an embedded µC can lead to the compromise of all embedded systems using that particular type of chip. However, an *embedded bootloader* is available on nearly all µCs and typically has full access to the chip's flash/RAM memories and its peripherals before loading the user application. Therefore, chip manufacturers integrate a security mechanism, which we refer to as CRP, in the bootloader to safeguard the integrity and secrecy of the firmware binary (and all cryptographic secrets and intellectual property within it). Because a variety of devices, ranging from automotive ECUs to IoT, often use the same or similar µCs, they also include the same, generic bootloader—with no control over its development and no insight into its source code. Hence, the users of µCs find themselves at the mercy of the quality of the chip manufacturer's internal security testing, if any such procedures are in place at all. Thus, however strong security primitives a specific system is built on, a vulnerable bootloader undoes all of this and makes the device susceptible to various attacks ranging from firmware readout to a full device compromise. Hence, bootloader security is of utmost importance for the integrity of the device and the secrecy of the firmware and the data within it. In notoriously secretive sectors such as the automotive industry, being able to extract firmware from ECUs allows for public scrutiny of the underlying security primitives, which are often of proprietary nature and insecure, as further shown in Chapters 4 and 6 and in [126, 125, 70].

The bootloader is the first piece of software that executes after reset and enforces the chip's CRP. Typically, it initialises essential peripherals (e.g., the internal clock) and loads and executes the application firmware. However, most bootloaders provide an external interface through a serial protocol, which typically uses an internal buffer to write and receive messages, making them prone to well-studied software vulnerabilities such as buffer overflows. These are aggravated by the lack of common mitigation techniques on embedded chips and especially in the bootloader, which often resides in a restricted memory area such as on-chip ROM and cannot be updated.

Hardware-based fault attacks induce a fault in on-chip computations, such as skipping an instruction, by changing the physical operating environment of the chip, e.g., the supply voltage. They do not rely on the presence of a software vulnerability. The literature covers a wide spectrum of hardware-based fault injection methods: the most widely-used techniques include voltage, optical, clock and electromagnetic fault injection. Optical fault attacks require extensive preparation such as decapsulating the chip [186], while electromagnetic fault attacks involve specialised hardware [49] and have a larger parameter space (e.g., probe positions). On the other hand, voltage fault injection ("glitching") does not require expensive lab equipment: open-source projects such as the Chipwhisperer [152] and the GIAnT [154] significantly lower the entry barrier for voltage glitching.

A large amount of research has focused on devising algorithm-specific techniques to recover keys when faults are injected into cryptographic computations (cf. for instance [208, 21, 18, 15, 96]). Such research usually assumes a specific *fault model* (e.g., a bit-flip in a certain part of a cipher's internal state), and ignores the details of how the faults actually influence the binary code implementing the cryptographic algorithm.

However, fault injection such as voltage glitching can often accurately target one particular instruction or memory location, and change the behaviour of normally secure code [135]. This makes bootloaders especially susceptible to said attacks: if for example the comparison instruction that checks if CRP is enabled can be manipulated, a single fault is sufficient to disable it. This in turn also compromises all cryptographic secrets stored on the μC, without the need for key recovery techniques specific to the implemented cipher.

Finding the correct fault injection parameters to “hit” a particular location (e.g., in the bootloader binary) is challenging in a real-world attack scenario: most published attacks treat the μC and its firmware as a black box, and thus have to resort to an (optimised) brute force search of the parameter space [24, 32, 158]. However, binary analysis of the bootloader binary could significantly reduce the search space. On the one hand, static analysis, which statically reconstructs the program control flow, reveals the possible bootloader execution paths to the CRP check. On the other hand, dynamic analysis, which leverages the bootloader execution, proves crucial in developing and testing CRP bypass exploits.

In this chapter, we propose several novel methods that bridge the gap between binary analysis and fault injection. We show that our approach enables complex attacks that would be infeasible without analysis of the bootloader binary.

## 5.2 Contributions

Combining software and hardware vulnerabilities, we apply binary analysis techniques on bootloaders of three different chips, which ultimately allow us to bypass their security mechanisms with inexpensive, open-designed hardware. Our approach is widely applicable and, unlike intrusive silicon-level attacks, scales well. Our research reveals several vulnerable design and implementation patterns in a bootloader that makes it vulnerable to attacks in this chapter and in the general literature. The contributions of this chapter are as follows:

- We extract and analyse four embedded bootloaders by three different manufacturers in detail. We show several software and hardware-based attacks on all bootloaders to bypass the CRP mechanisms. We perform all attacks with low-cost, open-design hardware, with a total cost of $\sim$ $ 250.

- We show how hardware-based fault injection through voltage glitching benefits from static and dynamic binary analysis techniques. Several novel attack methods arise from this approach, including the selection of glitch parameters based on the input-dependent execution path. Furthermore, dynamic glitch profiling on the STM8 ultimately leads to the—to our knowledge—first successful multi-glitch attack applied to a real-world target.

- We demonstrate that software exploitation techniques such as ROP, originally developed to bypass stack protection mechanisms for complex processors [176] and later extended to embedded applications [64], are also relevant for the bootloader security of simple, constrained µCs.

- We systematise the vulnerability classes identified in bootloaders and describe typical anti-patterns that need to be avoided in the development of secure embedded bootloaders. We explain these anti-patterns in Section 5.3 and reference them throughout this chapter as A1, A2, etc.

- All our tools, including the glitching hardware, will be made available as open source under a permissive license to aid the development of countermeasures and enable independent reproduction of our results[1].

**Attacker model** In this chapter, we consider the *hardware fault attacker* as introduced in [225], who can change the execution flow by introducing faults but cannot alter the bootloader binary. We assume that the adversary can obtain an identical, freely programmable chip or development board to profile the attacks and retrieve the bootloader code. Both assumptions are common in practice for embedded devices, where a physical attacker (aiming at firmware recovery) is often part of the threat model and where development kits for most µCs are readily available.

[1] Source available at `https://github.com/janvdherrewegen/bootl-attacks`

## 5.3 Secure Bootloader Design Directives

Bootloader vulnerabilities as presented in this chapter are not easily mitigated. Both chip manufacturers and their clients benefit from having the possibility to alter the flash memory content of the chip in case of malfunction or a firmware upgrade. The bootloader is the ideal candidate to incorporate this functionality. However, this requires a significantly larger codebase, potentially leading to software vulnerabilities such as described in Sections 3.3 and 5.4. Even though hardware fault injection attacks are hard to prevent, manufacturers can mitigate this risk by including extra components such as sensitive brownout detection [73] or a randomised internal clock [109] in the chip design. However, these mitigations undermine the overall performance of the chip and increase its costs, making a software-based approach—which would only affect the bootloader performance—attractive. Thus, in order to provide adequate reprogramming functionality without unnecessarily increasing the attack surface, we give several *anti-patterns* aimed at supporting the development of (more) secure bootloaders. These are design patterns we have observed both in our work as in previous research which *weaken* the protection mechanisms and thus must be *avoided* in an embedded bootloader.

**A1 - Partial RAM write access in protected state:** As shown in Section 5.4, µCs which have multiple protection levels often permit limited debug access to the chip's memory. Chips without an Memory Management Unit (MMU) must ensure that all bootloader memory and the area accessible to the user are separate. If no exploit mitigations are in place, a single compromise of the stack can jeopardise the whole system.

**A2 - Partial leakage of memory or registers:** Certain µCs still provide read access to RAM and/or registers when CRP is enabled. To exploit this issue, Obermaier et al. introduced cold boot stepping, which reconstructs the control flow of a program based on SRAM snapshots [149]. Furthermore, by single stepping a load instruction and manipulating CPU registers, Brosch recovers the firmware of a Bluetooth µC [26].

**A3 - Partial flash overwrite:** Having write access to one sector essentially gives an attacker read access to the chip. In addition to the attack described in Section 5.4.3, many systems have been broken by overwriting a flash sector with a program that reads out the entire memory of the chip [122, 69].

**A4 - Incomplete or non-atomic chip erase:** On many µCs, the CRP can be disabled through a full chip erase. However, as shown in [115, 76], in some cases that chip erase does not clear the full internal state (e.g., leaves RAM or data flash contents intact) and hence allows to recover information such as cryptographic keys stored in unerased memory. Furthermore, the erase process should be uninterruptible and atomic, that is to say that the bootloader should only disable the CRP at the very end of the erase process.

**A5 - Non-constant time code:** Timing leakage on password-protected bootloaders such as the Renesas M16C or TI MSP430 allows an attacker to recover the password byte-by-byte and gain access to the full flash memory [75, 160].

**A6 - Default to unprotected:** A comparison is easier to glitch if only specific value(s) *enable* the readout protection. For example, the LPC1343 bootloader starts with disabled protection unless a few specific values are read. Therefore, a much bigger range of glitches can cause this desired effect, e.g., if a load is forced to all zero or all one.

**A7 - Non-redundant check for readout protection:** On µCs without hardware countermeasures against fault injection, it is typically possible to bypass a single check with high success rate, e.g., through voltage glitching. However, as evident from Section 5.5, the success rate decreases exponentially with each redundant check.

**A8 - Large number of protection levels:** This can confuse developers as to exactly what kind of protection each level relates to. It is not uncommon for developers to use a manufacturer-provided IDE, which in turn could hide the low-level CRP details to the user and thus obscure which level is actually selected. Moreover, it may give the developer a false sense of security: if the chip does not adhere to other anti-patterns, such as A6, each security level requires exactly the same amount of effort to bypass (see e.g., [71]). However, we note that this is not inherently insecure, given the protection levels are correctly implemented and use.

**A9 - Separate On-Chip Debug (OCD) and CRP mechanisms:** On many µCs such as the Renesas V850, 78K0R and 78K0, or the TI MSP430, the readout protection mechanism is unrelated to the OCD access, which the user needs to either secure in software [167] or by blowing a fuse [129]. Due to this ambiguous setup, programmers can lose track of some ways to access the on-chip memory, ultimately undoing all other protection measures.

**A10 - Complex bootloader logic:** Every feature of the bootloader's communication protocol broadens the attack surface and thus entails more software exploitation risks. For instance, the USB storage emulation of certain LPC µCs, which contains a FAT filesystem implementation [145], could contain more issues, whereas the STM8 does not allow access to any bootloader commands if CRP is enabled. Besides, some Renesas µCs support up to three (UART, single-wire UART and SPI) communication interfaces in the same bootloader, which increases attack surface. On the flip side, a complex communication protocol such as USB would complicate dynamic and static analysis (cf. sections 5.5 and 5.6) of the bootloader binary, making the fault-injection parameter search harder.

In addition to the above anti-patterns, there are other possible approaches and tradeoffs to be taken into account for secure bootloader design. They include:

**Bootloader read protection** Some devices incorporate read protection of the bootloader memory space, preventing readout of the bootloader binary. For instance, recent chips may incorporate eXecute-Only-Memory (XOM), which utilises additional hardware to restrict a certain memory area (e.g., the bootloader section) to instruction fetches and disallows any read or write access. This would mitigate the attacks described in this chapter, because access to the bootloader binary is a prerequisite for each. However, Schink et al. analyse several XOM implementations in [174] and bypass the restrictions in each case to recover the protected code.

**In-field & field-return analysis** In certain scenarios, e.g., to perform dynamic in-field testing or to determine the cause of device failure, the manufacturer requires privileged access to the chip. This directly contradicts the proposed mitigations and anti-patterns, which are intended to lock down the chip as much as possible. Specifically, the manufacturer must balance anti-patterns A1 and A4 with leaving sufficient debug capabilities on the chip for these scenarios. An appropriate solution, though going against A9, would be to have a separate debug mechanism which only the manufacturer can access and is protected when the device goes into production.

## 5.4 Finding software vulnerabilities through static and dynamic analysis: NXP LPC1xxx bootloader

In this section, we study the bootloader of the LPC1343 [145] as an example of complex bootloaders containing software vulnerabilities (that do *not* require voltage glitching). We show how analysis of the bootloader binary is crucial for identifying and exploiting said issues, in particular a vulnerability that gives the adversary control over the stack and hence the program counter. We illustrate that the fixed memory layout facilitates exploitation with ROP [176, 64] techniques, which so far have received relatively little attention for embedded bootloaders.

The LPC1343 is from LPC1xxx family of NXP that encompasses a number of different chips based on an ARM Cortex M3 core. The results in this section likely generalise to other chips from this family as well. The LPC1343 bootloader implements two interfaces to access the chip's flash memory: (*i*) a character-oriented UART protocol and (*ii*) an emulation of a USB storage medium containing a single file representing the flash memory (cf. anti-pattern A10). We focus on UART, but note that the second interface also exposes substantial attack surface that we leave for future research.

The bootloader implements a CRP mechanism with five different access levels (cf. anti-pattern A8): NO_CRP, CRP 1, CRP 2, CRP 3, and an additional level called

NO_ISP [144]. These levels increasingly restrict the available bootloader commands: while NO_CRP gives full read/write access to the chip's memories, CRP 1 prevents read access to memory, restricts writes, and also disables the SWD interface. CRP 2 further restricts capabilities to essentially only a full chip erase, while CRP 3 permanently disables all programming functionality. NO_ISP disables the invocation of the bootloader, but leaves the SWD interface active, which can still be used to debug the processor and read memory (cf. anti-patterns A2, A9).

When CRP is enabled, the RAM region used by the bootloader (including the loaded CRP value) is not writable through the bootloader's "Write to RAM" command to protect against straightforward disabling of CRP. However, in this section, we show that the stack area in RAM is not protected on CRP 1, leading to a full bypass of the readout protection. During the responsible disclosure process, NXP confirmed that the vulnerability is present in all LPC1xxx series devices that do not incorporate a MMU. Concretely, based on the datasheets, we believe the following device series to be vulnerable: (*i*) LPC800 (*ii*) LPC1100 (*iii*) LPC1200 (*iv*) LPC1300 (*v*) LPC1500 (*vi*) LPC1700 (*vii*) LPC1800 (note that some LPC1500/1700/1800 feature an MMU).

### 5.4.1 Analysis of the LPC1xxx Bootloader

The bootloader resides in the 16 kB ROM from address `0x1FFF0000` to `0x1FFF4000`. We extracted the bootloader by transmitting that memory range over UART on a *profiling* device and loaded it into IDA Pro. We also used the bootloader's "Read from RAM" command for dynamically analysing the behaviour of the code when necessary.

The RAM memory layout (see Figure 5.1) (especially of the bootloader) deserves special attention for finding and exploiting potential software vulnerabilities. Therefore, it is described in detail in the following. We first discovered that the bootloader uses the lower part of the chip's RAM up to `0x10000300` as its working memory, as also mentioned in NXP's documentation [145], while it reserves the top 32 byte of the RAM for the flash programming commands. Addresses below the top 32 byte are allocated for the stack

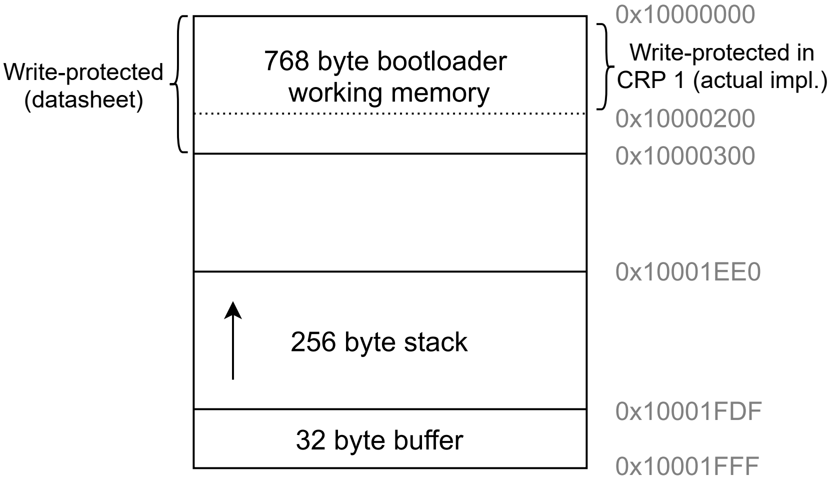

Write-protected (datasheet)
768 byte bootloader working memory
Write-protected in CRP 1 (actual impl.)
0x10000000
0x10000200
0x10000300
0x10001EE0
256 byte stack
0x10001FDF
32 byte buffer
0x10001FFF

Figure 5.1: RAM memory layout of the LPC1343 bootloader, indicating write-protected memory areas according to actual implementation and datasheet.

area. The stack grows towards lower addresses with a maximum size of 256 byte.

We confirmed that the CRP level is configured with a specific 32-bit value stored at address `0x2FC` in the chip's flash memory: `0x12345678` refers to CRP 1, `0x87654321` to CRP 2, and `0x43218765` to CRP 3, while other values leave the chip unprotected (*i.e.*, NO_CRP). Incidentally, this design anti-pattern A6 facilitates voltage glitching (cf. [71]), as we further discuss in Section 5.3. After reset, the bootloader loads the programmed CRP value from flash into RAM and uses the value in RAM for all subsequent CRP checks.

We then further statically analysed the bootloader commands, specifically the implementation of "Write to RAM" as shown in Listing 5.1. When the chip is configured in CRP 1, the "Write to RAM" command only proceeds if the target address is ≥ `0x10000200`, because memory below this address is used by the bootloader (and includes the buffered CRP level). Note that the start address of the RAM is loaded from a "hidden" configuration part of the flash as further described in [56].

Interestingly, while the manual [145, p. 329] claims that writes are permitted only above `0x10000300` and that the bootloader uses RAM up to to `0x1000025B`, the actual implementation permits writes above `0x10000200`, cf. Figure 5.1. We practically verified that we are able to write above this bound. We also noticed that the bootloader uses some variables located above 0x10000200 (specifically a pointer stored in 0x10000248 and referenced throughout the code). We have not further investigated whether this behaviour can be exploited, but it appears likely that this mismatch between specification and implementation could be misused.

Listing 5.1: Check in the LPC bootloader for RAM write range starting at offset 0x1fff0d94

```
ldr      r2, =0x438 // stores 0x10000000
ldr      r3, [sp, #0x28 + var_18]
ldr      r2, [r2]
adds     r2, #0xff
adds     r2, #0xff
adds     r2, #0x2   // add 0x200
cmp      r3, r2
// continue if address >= 0x10000200
bhs      continue_writing
// else set error code
movs     r4, #0x13
```

Crucially, we found that the implementation of "Write to RAM" does not protect the stack: there are no checks at all if the target address is on the stack (*i.e.*, $\geq$ 0x10001EE0). An attacker can exploit this to bypass the protection in CRP 1 and invoke the otherwise disabled "Read Memory" command, as further described in the following Section 5.4.2.

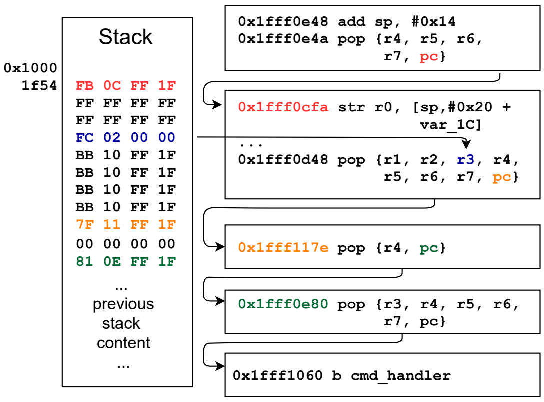
### 5.4.2 CRP 1 Bypass with Stack Overwrite

Because the stack is not write-protected, we can overwrite return addresses on it and hence control the program counter. We use ROP techniques to chain different gadgets in the bootloader to (*i*) jump into the "Read Memory" command and the (*ii*) jump back to the main command handler. Returning to the main command handler code prevents the bootloader from crashing and enables it to keep on receiving subsequent commands. First, we determined that the topmost return address on the stack is at `0x10001f54` while executing the "Write to RAM" command. We then write the following ROP chain, further illustrated in Figure 5.2, to the stack, starting at that address:

- `FB 0C FF 1F`: return address with a location behind the CRP check inside the "Read Memory" command handler (concretely, `0x1fff0cfa`[1]). This code then dumps 900 byte from a given starting address via the serial connection.
- When executed normally, the above code initially pushes seven registers (`R1`–`R7`) and ends in a corresponding `pop {r1, r2, r3, r4, r5, r6, r7, pc}` instruction. We therefore place seven 4-byte words on the stack. Crucially, the value later poped into `R3` controls the read target address (in the example ROP chain in Figure 5.2, this is set to `FC 02 00 00` to read the data starting at `0x2fc` in flash).
- `7F 11 FF 1F`: the control flow then returns to a gadget at `0x1fff117e`, which executes a `pop {r4, pc}` instruction. This is necessary to pad the overall stack layout (further explained below).
- This is followed by a 4-byte value to be poped into `R4`, and then the return address to proceed at:
- `81 0E FF 1F`: this is a `pop {r3, r4, r5, r6, r7, pc}` gadget. Due to the above padding, the remainder of the stack after this value is still in its original configuration and not overwritten, because it already contains valid values for `R3`–`R7` and a valid return address (in the main handler).

Applying this exploit repeatedly with different target addresses, we successfully read the complete flash and RAM within a few seconds. As the attack does not require voltage glitching, it can be carried out using a standard UART-USB cable, which is widely available for less than $ 5.

[1] Because the bootloader uses Thumb mode, all return addresses are incremented by 1 on the stack, so the written value is `0x1fff0cfb`



```
0x1fff0e48 add sp, #0x14
0x1fff0e4a pop {r4, r5, r6,
                r7, pc}

0x1fff0cfa str r0, [sp,#0x20 +
                     var_1C]
...
0x1fff0d48 pop {r1, r2, r3, r4,
                r5, r6, r7, pc}

0x1fff117e pop {r4, pc}

0x1fff0e80 pop {r3, r4, r5, r6,
                r7, pc}

0x1fff1060 b cmd_handler
```

Figure 5.2: ROP chain for bypassing readout protection of LPC bootloader and reading 900 byte from any start address (here: `0x2fc`). The exploit is applied by invoking the “Write to RAM” command as: `W 268443476 172`

### 5.4.3 CRP 1 Bypass with Partial Flash Overwrite

In addition to the above vulnerability, LPC1xxx devices also allow the partial erasure and overwrite of single flash sectors in CRP 1 (cf. anti-pattern A3). In an application note, NXP states that “though it is unlikely, it is conceivable that an attacker with knowledge about a system could partially overwrite firmware in such a way as to gain read access to internal flash memory” [144]. We confirmed that such an attack, akin to the methods demonstrated for PIC18F chips by [122, 69, 68], is indeed possible for the LPC1343 and other similar chips from the LPC1xxx family. At high level, the attack proceeds as follows:

1. The attacker overwrites one flash sector with dumper code padded by `nop` instructions, which outputs the contents of all other sectors e.g., through a UART port.

2. The attacker then resets the device, and the original initially runs as normal. When a jump or call references the overwritten sector, the dumper code is invoked.

3. The attacker then uses a second, identical device to overwrite a different sector with a similar dumper to recover the contents of the sector overwritten in step 2.

We practically verified that this method can be applied to the LCP1343 configured in CRP 1. However, this technique depends on the characteristics of the target program: it only works if the attacker overwrites a sector that contains code that is executed during the operation of the target program. A natural choice would be to use the boot sector, however, this is not possible in case of the LPC bootloader: CRP 1 prevents erasure and overwriting of sector 0. Therefore, the attacker has to potentially overwrite multiple sectors one by one until a sector that contains executed code is found. Furthermore, if only sector 0 contains active code, this method cannot be used to recover the flash contents, while the attack from Section 5.4.2 is independent of the target program.

### 5.4.4 Discussion

This section describes several attacks on chips configured in CRP 1. As mentioned before, the LPC1343 has five different access levels [144], with higher levels restricting read/write access to the chip further. For chips configured in CRP 2 and 3 specifically, the software-only attacks described in this section do not suffice. However, if an attacker can mount a successful fault injection attack, such as one described in the following sections, to downgrade the CRP to level 1, they again gain full access to the chip. In particular, by targeting the instruction which checks the 32-bit value indicating the CRP (cf. section 5.4.1), an attacker may force the chip into a lower protection level. Indeed, Gerlinsky uses voltage glitching to exploit anti-pattern A6 in [71] to bypass the CRP check altogether. This hardware fault attack unlocks the chip in any access level, and thus breaks its security regardless of its CRP configuration.

## 5.5 Glitching guided by dynamic analysis: The STM8 bootloader

ST's STM8 series chips feature a bootloader which provides the user with read, write and erase functionality [191]. It incorporates a readout protection mechanism to prevent an attacker from connecting to the bootloader. In this Section, we analyse the embedded bootloaders of two STM8 chips, namely the STM8L152C6 [194] and the automotive STM8AF6266 [193], the latter being used in car immobilisers, a security sensitive anti-theft component (cf. Chapter 6). As shown in [169], where three researchers each spent three months voltage glitching the bootloader of an STM32F2 chip before they eventually succeeded, finding the correct glitch parameters to bypass readout protection is far from trivial.

The lack of any feedback when voltage glitching the STM8 bootloader makes it an even more challenging feat. Unlike other bootloaders (cf. e.g., Sections 5.4 and 5.6), which allow certain commands and only restrict security-sensitive functionality such as flash reads and writes, the CRP on the STM8 prevents any communication with the bootloader if enabled. Hence, until the bootloader activates the serial interface and thus pulls the UART receive pin high, we are completely in the dark as to what the injected glitch has achieved. An attacker could employ power analysis or other side channels to determine differences in instruction flow, however this typically requires many traces. Therefore, we use a technique to facilitate bootloader glitching, which we call *bootloader grey-box glitching*. By flashing security critical parts of the bootloader as a user application, we gain an invaluable advantage to profile the glitches individually. We then use that information to set up a complex double-glitch attack.

Because the bootloader is shared among all chips of the same version and likely more (according to [191], there are only four versions of the STM8 bootloader, whereas there are dozens of different STM8 chips), knowledge of its operation is invaluable when glitching other similar chips.

**Attack assumptions** We assume that the chip under attack is programmed (e.g., the first flash byte equals `AC` or `80`) and has the highest level of protection enabled (e.g., the bootloader is disabled and readout protection is active). We would like to emphasise that even though we only demonstrate this attack on the STM8AF6266 and STM8L152C6, we believe the methodology would generalize to other STM8 chips with a different bootloader version.

### 5.5.1 Bootloader Extraction and Analysis

We obtained the two bootloaders of the analysed STM8 chips by connecting to the on-chip Serial Wire Interface Module (SWIM) [192] debug interface of a profiling device and issuing a read command for the address range `0x6000`–`0x8000`, which is the bootloader ROM according to the datasheets. The CRP on the STM8 works as follows: two option bytes, namely the CRP[1] and Bootloader Enable (BL) bytes stored in EEPROM control whether the bootloader activates or loads and executes the application code. If either the chip is empty (according to the datasheet, an STM8 chip is deemed empty if the flash byte on address `0x8000` does not equal `0x82` or `0xAC`) or the BL byte is set to a certain value[2], the bootloader continues to check the CRP byte. Finally, if the CRP byte indicates readout protection is disabled, the bootloader activates its serial interfaces (it supports both UART and SPI) and waits for further programming commands. From here on, we will refer to this part of the bootloader as the *serial bootloader*. The serial bootloader performs no further CRP checks and thus grants full read and write access to the firmware, making the STM8 a suitable target for voltage glitching attacks.

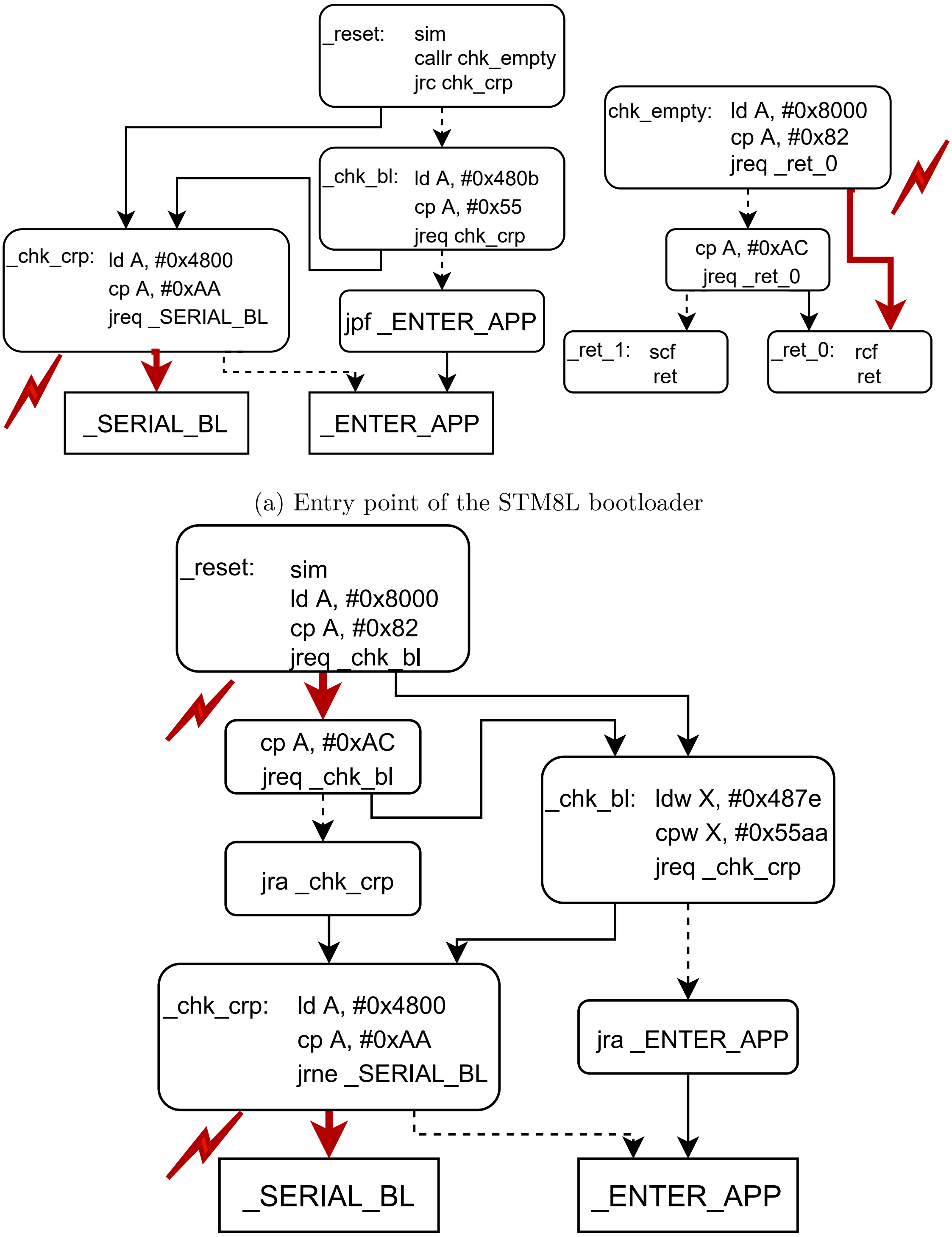
Figure 5.3 depicts the control flow diagrams following the entry point of both the STM8L152C6 and STM8AF6266 bootloaders. If the µC is blank, or the BL option byte is set to `55(AA)`, it proceeds to check the CRP byte. The CRP on the STM8L152 is disabled if this byte equals `0xAA`, and only then the serial bootloader activates. On the STM8AF6266, readout protection is only enabled if the CRP byte equals `0xAA` (cf. anti-pattern A6).

### 5.5.2 Profiling Critical Bootloader Sections

Glitching a certain instruction on a µC often requires precision in the range of *nanoseconds*. Thus, even for a single fault, exhaustively searching the entire glitch parameter

[1] ST refer to this byte as `ROP`, however to avoid any confusion with Return-Oriented Programming, we will call it CRP here

[2] cf. datasheet of the particular STM8 chip for the exact value of the option bytes



(a) Entry point of the STM8L bootloader

(b) Entry point of the STM8A bootloader

Figure 5.3: Control flow diagrams of the STM8L and STM8A bootloaders. Jumps are displayed with a full line, whereas a dotted line implies a fallthrough path. Glitch paths for each µC with first flash byte 82 are indicated in red.

space—consisting of the glitch offset $T$, width $W$ and voltage $V_F$ as well as the normal operating voltage $V_{CC}$—quickly leads to a state explosion. The STM8 incorporates Brownout Reset (BOR) circuitry holding the chip under reset when $V_{CC}$ drops below a user specified threshold (which they can set through an option byte in EEPROM). This threshold can range from 1.8 V to 3 V, so in order to circumvent the BOR circuit altogether, we keep the normal operating voltage at 3.3 V, leaving us with three unknown parameters. The goal of this phase is to optimise the glitch voltage and width by temporarily minimizing its timing aspect.

### Critical Bootloader Sections

First, we define a Critical Bootloader Section (CBS) as a logically coherent unit of basic blocks which either check the BL or CRP option byte, or directly precede the serial bootloader. The STM8 bootloader checks both option bytes at the very beginning of its execution and does not continue unless BL is set and CRP is disabled. Thus, by design, identifying the critical sections on this µC does not require extensive reverse engineering.

Next, to *separately* run and fault each CBS on the real µC hardware with dynamic analysis capabilities, we insert the CBSs one by one into the custom user application stub depicted in Listing 5.2 and flash it as a normal user application onto the µC. On chips which are difficult to fault (e.g., the glitch only succeeds if both $V_F$ and $W$ lie within a narrow range which faults the instruction but does not reset the µC), this technique significantly reduces the parameter space. The stub pulls a GPIO pin high before executing the specific bootloader section on which we trigger the glitch, and indicates success by pulling a different GPIO pin high. We define success as reaching a basic block which the application would never enter in normal operation. As shown in Listing 5.2, a successful glitch in the `check_empty` section would result in the `scf` instruction being reached (which has been replaced in the template), indicating that the chip is empty and thus making the bootloader progress into the `check_CRP` section.

Listing 5.2: Flashing the check_empty CBS enabling the search for voltage and width parameters

```
    PE_ODR |= 0x80 // generate trigger
chk_empty:
    ld A, #0x8000
    cp A, #0x82
    jreq _ret_0
    cp A, #0xAC
    jreq _ret_0
_ret_1:
    PE_ODR |= 0x01 // indicate success
_ret_0:
    ret
```

#### Reduced glitch parameter search

Intuitively, since the CBS executes immediately after the trigger, the offset search space reduces significantly. This allows us to essentially focus on the glitch voltage and width, which are mutually dependent: a deeper glitch, for example to $V_F = 0.5\,\text{V}$, must be very short ($W \approx 50\,\text{ns}$) in order not to reset the chip. Figure 5.4 shows the various width/voltage pairs which produce a successful glitch in the _reset block in the STM8A bootloader. Additionally, in this phase we get a rough estimate of the *glitch success rate*, however, due to other influences such as the glitch timing, we do not achieve the same rate when glitching the complete bootloader, as shown in Section 5.5.3.

### 5.5.3 Partially Attacking the Bootloader on Reset

In the second phase we move on to glitch the real bootloader, which poses several additional challenges. Firstly, even though the STM8 datasheet states that the µC restarts

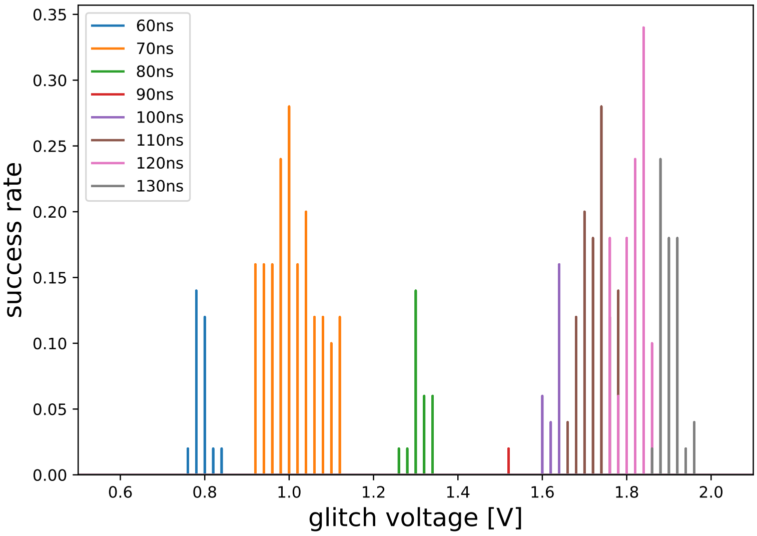


Figure 5.4: Success rate for glitching the *_enter_app* bootloader section (which immediately precedes the serial bootloader) on the STM8A at a constant offset of 0.34 µs

with an internal 2 MHz clock, we have noticed that the bootloader writes the Clock Master Divider Register (`CK_DIVR`), which controls the CPU frequency, just before loading the user application. Hence, we still have to determine the actual reset frequency of the bootloader. In a similar fashion to [169], we achieved this by connecting a 30 Ω shunt resistor between the $V_{ss}$ pin and the ground, in essence lifting the chip's ground. This allows us to measure the power consumption of the chip, which reveals the real clock frequency of the µC after reset as shown in Figure 5.5. A second issue arises due to the built-in Power-On Reset (POR) circuit, which generates a reset signal when the chip powers up. Hence, the bootloader does not execute immediately once we pull the reset pin high. Again, as shown in Figure 5.5, the power consumption gives us an estimate for when the bootloader starts operating and thus reduces the offset search space significantly. With the glitch voltage and width set to the values acquired in the first phase, we can now scan the offset search space. Before moving on to a fully locked chip, we set the `BL` and `CRP` option bytes individually, such that we only need one glitch to reach to the serial bootloader.

| **CRP** | **BL** | [8000] | **section** | **T** [µs ] | **success rate** [%] |
|---|---|---|---|---|---|
| `AA` | `00` | `82` | `chk_empty` | **29.5** | **0.6** |
| | | | `_chk_BL` | 35.75 | 0.11 |
| | | `AC` | `chk_empty` | **30.5** | **0.5** |
| | | | `_chk_BL` | 36.25 | 0.1 |
| | | | | 36.75 | 0.1 |
| `00` | `55` | `82` | `_chk_CRP` | **38** | **0.6** |
| | | `AC` | | **39** | **0.5** |

Table 5.1: Glitch success rate and their offsets triggered on reset of the critical bootloader sections of the STM8L. Glitch voltage and width kept constant at $V_F = 1.84\,\mathrm{V}$ and $W = 50\,\mathrm{ns}$

### Comparison: STM8L vs. STM8A

Table 5.1 gives an overview of the various offsets leading to the serial bootloader on the STM8L with all possible combinations of the first flash byte and one option byte set. All glitches are aligned with either the rising or falling edge of a 2 MHz clock from reset, which can be attributed to a stable internal oscillator. In order to achieve a success rate above 0.1%, the glitches need to fall within the vicinity of 20 ns of the given offsets, proving the necessity of the earlier profiling phase. The bootloader code clarifies and helps predicting the glitch timing for different bytes on address `0x8000`. For instance, if the first flash byte equals `0xAC`, the glitch on the CRP byte check falls 1 µs, or two clock ticks, later due to the execution of an additional basic block in the `check_empty` subroutine. As to be expected due to a different internal design, performing the same experiment on the STM8A does not yield exactly the same result. Firstly, the reset time, e.g., the time it takes for the bootloader to start executing, on the STM8 is roughly 78 µs, opposed to 26 µs on the STM8L. In addition, the internal oscillator on the STM8A does not seem to be as stable as its STM8L counterpart, making the glitch offsets fall more within a range of $\sim$ 6 µs: glitches in the `_reset` section fall in the range of 79.67 µs–86.56 µs, whereas glitching the `_check_crp` section is most successful in the 86.68–93.54 µs range.

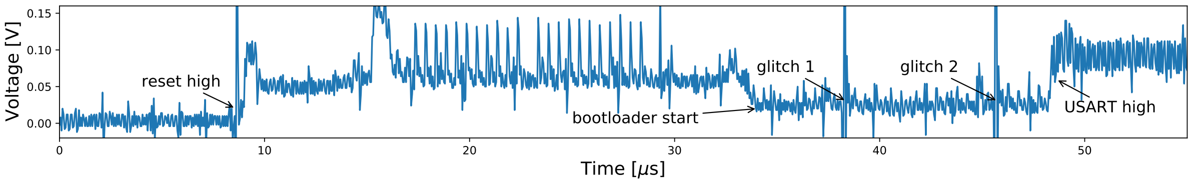


Figure 5.5: Power consumption of the STM8L152 upon reset measured with a 30 Ω shunt resistor.

### 5.5.4 Full Double-Glitch Attack

Finally, once we have the correct parameters for the separate glitches, we have to combine them to attack a fully locked chip. The main challenge is that we receive no feedback until the `USART_RX` pin is pulled high indicating the activation of the serial bootloader. Table 5.2 gives the final glitch parameters for the chips we have investigated. What stands out is that the success rate for the double-glitch attack on the STM8L is substantially lower than 0.0036%, which is what we would expect when combining both individual glitch success rates. We attribute this peculiarity to the 3-stage pipeline on the STM8: the first glitch makes the bootloader jump directly to the `_chk_crp` section after returning from `chk_empty`, whereas in Section 5.5.2 the bootloader goes through the `_chk_bl` section first. Thus, the pipeline content differs at the point of the `_chk_crp` glitch in both scenarios. We repeated the partial attack for the µC with an 'empty' chip (e.g., with `00` as first flash byte) and CRP enabled, and indeed, the individual glitch success rate for the `_chk_crp` section in this scenario is 0.02%. For reference, 100k glitches, the average number of attempts needed to bypass the CRP on STM8AF6266, takes ∼ 2.5 min on our setup. Since the CRP check only occurs once at boot time, we reset the chip for each glitch attempt. Given the final success rates for the double glitch attacks, we would like to emphasise the difficulty of a black-box only approach (e.g., without inspecting the bootloader code) to glitch the CRP check. Without knowledge of the execution path and time between both target checks, a full search of the glitch parameters quickly leads to a state explosion. The lack of feedback until both glitches succeed only aggravates this, making a strong argument for our greybox approach.

| **chip** | $T_0$ [µs ] | $W_0$ **[ns]** | $T_1$ [µs ] | $W_1$ **[ns]** | **success rate** [%] |
|---|---|---|---|---|---|
| STM8L152C6 | 29.5 | 50 | 7.32 | 50 | 0.0001 |
| STM8AF6266 | 80.75 | 120 | 3.91 | 120 | 0.001 |

Table 5.2: Glitch parameters for fully locked STM8 chips with first flash byte `82` triggered on reset.

## 5.6 Glitching guided by static analysis: Renesas 78K0 bootloader

From a program analysis perspective, the STM8 bootloaders described in Section 5.5 are fairly straightforward to analyse, because their CBS executes immediately after reset, and does not continue unless protection is disabled. In contrast, many other bootloaders allow basic functionality and only restrict certain security sensitive commands if CRP is active.

Intuitively, glitching such bootloaders poses a number of different challenges. First, the communication protocol and bootloader code are often intertwined, making the identification of CBS (*i.e.*, where we want to inject a glitch) non-trivial. Moreover, different commands typically have their own unique handler code, making the glitch offset dependent on the specific handler and the content of the command, among others. In this section, we leverage symbolic execution to predict glitch offsets based on the execution path taken in the command handler code of the Renesas 78K0 bootloader. The 78K0 is a multi-purpose 8-bit low-power Renesas microcontroller which, similar to the STM8, is often used as the central µC on certain automotive immobiliser systems as further shown in Chapter 6. We evaluate the effectiveness of our white-box technique and compare it to the black-box attack by Bozzato et al. [24].

### 5.6.1 78K0 Bootloader Extraction and Analysis

An external programmer can activate the bootloader on Renesas µCs by pulling the `FLMD0` pin high on reset. Subsequently, it can select either SPI or UART communication modes to interface with the on-chip bootloader [166]. The user can increase the level of security by clearing individual bits in the security byte, which respectively turn on write, block erase, chip erase and boot sector write protection on the 78K0 (cf. anti-patterns A8 and A9).

Regardless of the security configuration, the bootloader always allows executing certain commands such as `verify` or `checksum`, which confirm and calculate a simple checksum over blocks of 256 byte respectively. Other commands, like `program` or `erase` only succeed if the respective security bit is `1`. All flash-related bootloader commands take a 3 byte start and end address as argument. With the 78K0 being an 8-bit μC, arithmetic on these 3 byte addresses is performed byte-wise.

The bootloader region is not mapped to memory during regular operation and thus cannot be read from a normal application. However, Renesas provides a "flash self programming library", which users can include in their application to interface with on-chip firmware that performs flash operations [168]. The library function `FlashInit` sets the μC into flash programming mode by first writing `0xA5` to the `FLMDPCMD` register, which enables the writing of flash-specific registers. Next, it enables the flash programming mode by writing to the `FLMDMCR` register, which consequently maps the bootloader and on-chip firmware in a memory region marked as "reserved" in the datasheet.

### 78K0 Command Handlers

The reset vector of the bootloader continues with the bootloader entry point, which in turn progresses into the main command processing loop. Each serial command is identified by a specific command byte, all of which are stored in an array at address `0x544`, adjacent to an equally sized array of pointers to their respective handler code. All flash-related bootloader commands take a 3 byte start and end address as argument. With the 78K0 being an 8-bit μC, arithmetic on these 3 byte addresses is performed bytewise. Consequently, depending on the results of these byte level comparisons, small timing differences appear, which affect the glitch offset. Thus, in order to accurately predict the glitch timing, we need to map each address pair provided in the command buffer to an execution path in the command handler code.

### Attack strategy

Bozzato et al. propose three attacks to read out the full firmware on a 78K0 by exploiting several bootloader commands [24]. A first attack glitches the `checksum` and `verify` commands to operate on 4 bytes instead of the minimum allowed 256 bytes, which allows an attacker to guess 4 bytes of firmware per successful glitch. They base the guess on a different glitch in the `checksum` command, which can leak bytes individually though is not completely accurate (e.g., the address or value of the leaked byte can be wrong). This is the only known attack on this µC which preserves the original firmware completely. In the two other attacks, they exploit the `erase` and `program` commands to overwrite part of the boot section with a dumper program (cf. anti-pattern A3). Since all of these commands require a start and end address to operate on, all handlers initially call the same sanity check function. In contrast to Bozzato et al., who employ a genetic algorithm to determine the best glitch offset in a black-box manner [24], we make use of the full knowledge of the bootloader binary.

## 5.6.2 Constraint-based Glitching

Symbolic execution is a widely used technique in software testing and program analysis [106]. A symbolic executor tracks constraints over the range of values *symbolised* input variables can take along an execution path. It can use the constraints to generate a viable input value that will cause the execution of that path in a concrete execution. Logically, these constraints can also be used to verify if a given input value will cause the execution of a particular path. We leverage the latter technique to statically create classes of arguments which follow the same execution path through the targeted handler code, and thus will result in the same glitch offset. Our technique operates on the assembly language instructions, as opposed to lifting to an intermediate representation. Unfortunately, state-of-the-art symbolic execution engines such as Angr [178] and KLEE [29] do not provide out-of-the-box support for exotic architectures like the 78K0. However, the main reason for this decision is to retain low-level information such as instruction cycles. This proves crucial in predicting and classifying offsets for hardware-level attacks such as voltage glitching.

### Constructing argument equivalence classes

Our framework uses the bootloader control flow graph to statically calculate the constraints along all paths through a certain command handler. The only prerequisites for our technique are: (*i*) Extraction of the bootloader control flow graph from a disassembler such as IDA Pro [88] or Ghidra [209]: the auto-analysis is usually adequate for simple serial protocols. (*ii*) Identification of the targeted bootloader command handler code: since the bootloader communicates through a serial interface, it typically suffices to follow the corresponding serial interrupt handler to find this. The extensive use of constants (*i.e.*, error codes) can further help locate handler code. We then perform a depth-first search from the command handler entry point to the target instruction. Since we are interested in the constraints along the complete path, we recursively repeat this process for all calls along the path. We mark the input variables to the command handler (cf. Appendix B) as symbolic and record their constraints along each path. We propagate the arguments for each conditional branch to check whether they originate from the initial input variables. Then, we use a simple constraint solver (*i.e.*, `python-constraint`) to obtain all viable paths through the command handler and their respective constraints. We then define an argument's equivalence class within the handler as follows:

**Definition 5.** Given a function $f$ with input arguments $A_n, ..., A_0$, we define an *equivalence class* on this function as the set of all arguments which result in the same execution path through the function.

Finally, we use instruction cycle count information from the datasheet to calculate the number of cycles a path, each corresponding to one argument equivalence class, takes.

**Constraint limitations** Our constraint-solving approach targets simple protocols and does not handle any cryptographic operations or intricate constraints. Since the bootloader memory is generally limited in size, they are typically not of very complex nature. Our approach tracks and solves simple constraints (e.g., a comparison with a constant or other symbolic variable) appearing along the execution paths in the bootloader. In its current state, it does not propagate taint or handle complex constraints. However, this functionality can be added at a later time should this be required. Since this technique is aimed at narrowing down the glitch offset search window, it does not strive for completeness (*i.e.*, catch all possible constraints on the path). It suffices to solve all constraints on variables directly influencing the glitch offset, which we do by symbolising the input buffers to the command handlers. Another issue arises when handling a path containing code loops which depend on a dynamically set variable. Since our static technique cannot determine how many iterations the loop goes through, we detect the loop and output the cycle count of the loop body. When glitching, we can empirically test the number of iterations by adding a multiple of the loop cycle count to the glitch offset each time, until we hit a successful glitch.

### Practical appplication on the 78K0 bootloader

A similar pattern emerges in the handler code for all flash related commands: at the very start, each handler calls a function, shown in Listing 5.3, which processes both addresses provided in the command buffer. Concretely, it calculates which 1 KiB flash block each address resides in and performs certain sanity checks on the arguments, e.g., if the end address falls within the on-chip flash range and whether the start address is lower than the end address.

Listing 5.3: Pseudocode of the `sanity_check` function, with arguments $A_0$ and $A_1$ the start and end address provided in the command buffer. $A_{max}$ indicates the highest allowed flash address on the chip.

```
int sanity_check($A_0$, $A_1$) {
    $b_0$ = get_block_no($A_0$);
    $b_1$ = get_block_no($A_1$);
    if (cmp_addr($A_0$, $A_{max}$) > 0)
        return −1;
    if (cmp_addr($A_0$, $A_1$) > 0)
        return −1;
    return 0;
}
```

Figure 5.6 depicts a simplified control flow graph of the function that calculates the block number for a given flash address, called twice from within `sanity_check`. If the address is non-trivial (*i.e.*, not `0` or `ffff`), the code calculates the block number by merging the two least significant bytes into two 8-bit registers, and dividing these by `0x8` and `0x80` consecutively. Unsurprisingly, a 16-bit division on an 8-bit µC takes considerably longer than any other operation: on the 78K architecture, `udivw` completes after 25 clock cycles, whereas a simple `cmp` takes 4 cycles. The `cmp_addr` function follows a similar pattern but results in a smaller execution time difference, because it only compares its arguments byte by byte, starting from the most significant byte, and thus does not include any overly time-consuming operations such as division. For reference, we include the full assembly code in Appendix A.

### 5.6.3 Exploitation and Evaluation

In both the `checksum` and `verify` handlers, the targeted length check takes place directly after the code returns from the `sanity_check` function. Thus, applying our symbolic execution technique to each command handler yields nine sets of arguments (with $A_1 = A_0 + 3$, the smallest possible range on the 78K0) that result in a distinct execution path. Figure 5.7 shows the actual glitch offsets of each of those sets and reveals the main advantage of our technique compared to black-box predictions of a genetic algorithm: by

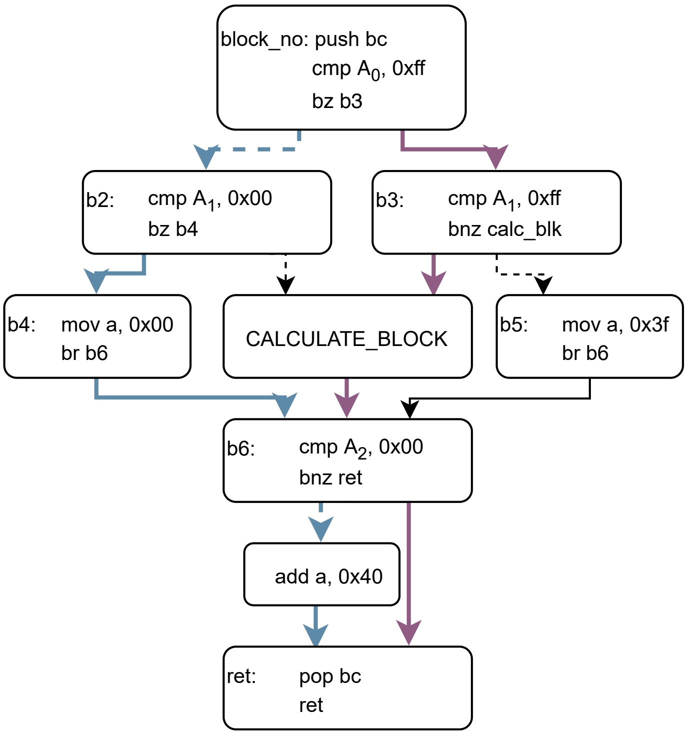


Figure 5.6: Simplified control flow graph of the `get_block_no` function, which takes as input a 3 byte address $A = A_2A_1A_0$. To illustrate, we depict example execution paths for the equivalence classes of `0x1004c` and 0x5ff

.

basing the glitch offsets on execution paths, we can accurately predict glitch offsets of other sets of addresses by taking the execution cycles of each path into account. Table 5.3

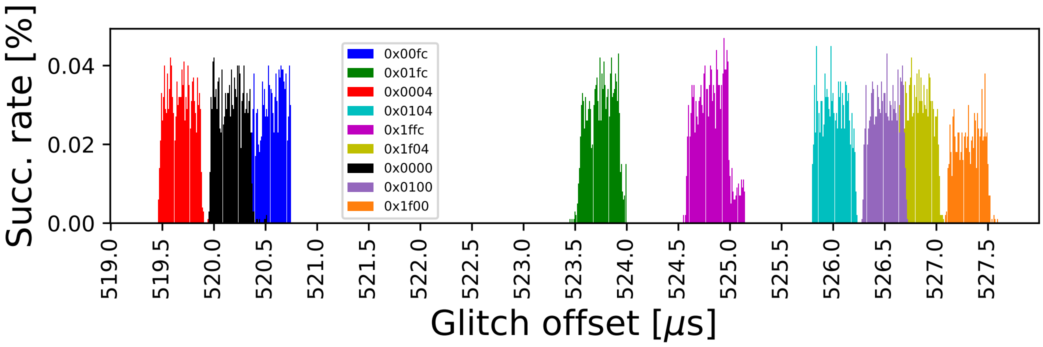


Figure 5.7: Glitch success rate and offset of the `checksum` command for the first address of each equivalence set. Glitch triggered on falling edge of the SPI clock pin on a 78K0/KC2 µC powered at 3 V clocked by 8 MHz internal oscillator. Glitch voltage and width constant at 0 V and 120 ns, respectively.

compares the cycle count for each equivalence class path through the `checksum` command handler obtained by our technique with the actual glitch offsets. Our technique calculates all viable execution paths from the start of the command handler to the basic block indicating a parameter error. Firstly, we note that even though in absolute terms the offset difference between classes is less than what we would expect from the cycle difference (e.g., a difference of 12 cycles between `0004` and `0000` translates into a real offset difference of $0.5\mu s$ instead of an expected $1.5\mu s$ @ 8 MHz). This could be attributed to the actual internal clock frequency, which depends on the input voltage, ambient temperature and the manufacturing tolerances of the particular chip. However, taking this discrepancy into account, the deltas of the real offsets and the calculated cycles are consistent. Only paths with the least significant byte of the end address equal to `ff` diverge from this. We attribute this to the glitch occurring on a different instruction to the other paths, due to there being a specific check for this byte and thus facilitating the path to the target basic block.

We compare our technique to the genetic algorithm published by Bozzato et al. in [24]. They inject arbitrary glitch waveforms and tailor a genetic algorithm to optimise both

| **equiv. class** | **cycles (c)** | $T[\mu s]$ | $\frac{\Delta T}{\Delta c}$ | **equiv. class** | **cycles (c)** | $T[\mu s]$ | $\frac{\Delta T}{\Delta c}$ |
|---|---|---|---|---|---|---|---|
| `0004` | 454 | 519.49 | n/a | `0104` | 610 | 525.82 | 0.041 |
| `0000` | 466 | 519.97 | 0.040 | `0100` | 624 | 526.32 | 0.040 |
| `00fc` | 536 | 520.39 | 0.011 | `1f04` | 632 | 526.65 | 0.040 |
| `01fc` | 614 | 523.57 | 0.025 | `1f00` | 644 | 527.44 | 0.042 |
| `1ffc` | 642 | 524.63 | 0.027 | | | | |

Table 5.3: Comparison of predicted number of cycles and actual glitch offset (first offset with a glitch success rate above 3%) for each equivalence class for the `checksum` command. Differences ($\frac{\Delta T}{\Delta c} = \frac{T_n - T_0}{c_n - c_0}$) are calculated from the offset $T_0$ of the first encountered equivalence class, `0004`

the glitch offset and shape. As no source code is available for their approach, we have to rely on the published performance characteristics, which include results for two bootloader commands, namely `checksum` and `verify`. Table 5.4 summarises the results of the comparison of our technique with the arbitrary waveform injection and a more generic pulse injection, which resembles the GIAnT as used in our work. While we completely focus on the timing aspect of the glitches, and leave the actual shape, voltage and glitch width constant at $V_F = 0\,V$ and $W = 120\,\text{ns}$, our technique performs better than the black-box approach with a normal pulse style glitch.

| **method** | `checksum` | `verify` |
|---|---|---|
| `wvf` [24] | 4.2% | 6.8% |
| `pulse` [24] | 2.8% | 3.7% |
| `equiv` [here] | 3.2% | 4.3% |

Table 5.4: Success rates of different glitch search strategies on `checksum` and `verify` commands. `wvf` and `pulse` use the same genetic algorithm, with the latter using a pulse-shaped glitch similar to our hardware, whereas the former also optimises the glitch waveform.

As for the arbitrary waveform injection, as shown in Figure 5.7, our success rate only surpasses 4.2% on 2% of the offsets, which appears to be the upper limit for our constant voltage and width. Additionally, because the pattern shown in Figure 5.7 emerges in all other flash-related handlers, glitching another command does not require searching for the parameters again. Assuming a 5 % success rate for guessing a byte (we have not included this glitch since it is very similar to the described `checksum` attack), extracting the firmware of an 8 kB chip would take $\sim$ 10 hours.

**Discussion and limitations** We would like to emphasise that, even though we only demonstrate it on the 78K0 bootloader, this technique applies to many other µCs as well. In a scenario where either the glitch offset is close to the trigger (e.g., in the range µs), or where we can match a code section in the bootloader to an externally observable anomaly (e.g., a faulty serial message checksum, which is always calculated on the contents of the message), we can compute the code paths to the targeted area. In cases where an initial search of the parameter space does not yield any positive results, or where the bootloader is hardened against glitching attacks (e.g., by performing redundant checks), a more fine-grained search is required. By statically calculating the possible execution flows from a set point to a targeted section, we can reduce the offset search space and shift our focus to other glitch parameters. However, for glitch offsets which occur too long after a trigger, this technique can lead to a state explosion and will require heuristics to reduce the number of possible paths. Finally, because we focus on the glitch timing, a combination of our technique to accurately predict the offset and a different strategy to optimise the remaining parameters might yield even better results.

## 5.7 Dynamic vs Static Approach

In this chapter, we have shown how a preliminary analysis of the bootloader binary can significantly enhance the chances of mounting a successful fault attack. It is very challenging to exactly pinpoint why certain glitch parameters produce a successful hardware fault attack without delving deep into the physics on the silicon level. Therefore, many techniques in the literature have proven successful in different scenarios. Moreover, even though the approaches described in sections 5.5 and 5.6 diverge significantly, they both show how isolating a certain area of the parameter search is essential for improving the success rate of a glitch attack. Depending on the scenario, an attacker may want to choose either one, or opt for a combination of both techniques.

The technique described in section 5.5 performs well in a scenario where the timing aspect of the glitch is clear, but glitching a particular instruction proves challenging. Typically, chips which perform a CRP check at boot time, such as the STM32 [169] and STM8 (cf. section 5.5), are good candidates for this approach. Generally, power analysis can accurately reveal when the chip loads and checks the CRP [169, 71], which facilitates the glitch timing. However, if the particular instruction is hard to glitch (e.g., the glitch only succeeds if the voltage and width fall within a narrow range), an attacker can apply the dynamic approach detailed in section 5.5 to optimise the remaining parameters.

In contrast, the static approach, as described in section 5.6, targets a slightly different scenario. That is, where the timing of the injected glitch matters most, but the chip allows a wide range of glitch shapes (e.g., an attacker can set the glitch voltage to 0V and the width within the vicinity of a clock tick). This scenario typically arises in bootloaders which allow the execution of certain commands (e.g., requesting a firmware checksum), but prohibit other security sensitive commands, such as a read/write from/to memory. In that case, a static analysis of the paths taken through the bootloader binary can significantly narrow down the search space for the timing aspect of the attack. This technique is especially valuable in a scenario which requires a number of glitches in the range of millions (*i.e.*, each successful glitch only leaks a few bytes of the firmware), as shown in section 5.6 and [25].

Finally, in the most intricate scenario, where all of the glitch offset, width and voltage are challenging to find, a combination of the aforementioned techniques would prove most efficient. A preliminary dynamic analysis on a profiling device, akin to the technique described in section 5.5 narrows down the glitch shape. Equally, building up the constraints along the paths through the bootloader binary (cf. section 5.6) cuts down the offset search space. Then, this reduced parameter space should allow an attacker to mount a successful fault-injection attack.

| Technique | Example(s) | Scenario |
|---|---|---|
| blackbox | [25, 32] | Meaningful feedback for classifying glitch outcome |
| power analysis | [71, 169] | Narrows down $T$ - often in combination with A6 |
| dynamic analysis | [section 5.5] | $T$ clear but small successful range for $W$ and $V_f$ |
| static analysis | [section 5.6] | Wide $W$ & $V_f$ range but requires exact $T$ |

Table 5.5: Overview of the effectiveness of voltage fault-injection techniques in different scenarios. We include a non-exhaustive list of references for each technique

To summarise, table 5.5 provides an overview of the scenarios in which techniques in the literature and this chapter are most effective.

## 5.8 Chapter Summary

Voltage fault injection is a powerful technique that has been widely studied in the context of cryptographic primitives [208, 21, 18], but comparatively little research has been done in the context of traditional software security. In this chapter, we have brought advancements in binary analysis such as symbolic execution and dynamic analysis into the low-level hardware security domain. Furthermore, we also show that exploitation techniques like ROP apply in the context of embedded bootloaders.

We demonstrated that symbolic execution can be leveraged to define argument equivalence classes based on their respective execution paths. This gives us valuable insights into their execution times, which allows us to produce precisely-targeted glitch offsets to aid the glitch parameter search as we demonstrated on the 78K0 bootloader. By flashing parts of the bootloader as application code and thus enabling dynamic glitch profiling on the target hardware itself, we have presented here the first fully documented multi-glitch attack against the widely used STM8 family of microcontrollers, which gives full access to the device's memory. The techniques presented in this chapter are applicable to other families of microcontrollers and can be fully implemented using inexpensive open-designed hardware.

CHAPTER 6

# SECURITY ANALYSIS OF DST80 IMMOBILISER ECUS

## Contents

In this chapter, we perform a security analysis of DST80 immobilisers of two different manufacturers. We apply and improve on techniques presented in Chapter 5 to retrieve the immobiliser firmware, from which we reverse engineer the proprietary DST80 cipher, which we present here in full detail. Next, we reveal the use of weak key diversification in the immobiliser firmware, leading to full key recovery. We also point out several configuration issues in DST80 transponders. This chapter is based on the following publication:

*Wouters, L., Van den Herrewegen, J., Garcia, F. D., Oswald, D., Gierlichs, B., & Preneel, B. (2020). Dismantling DST80-based Immobiliser Systems. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2020(2), 99-127.*

## 6.1 Motivation

A vehicle immobiliser system is an imperative anti-theft device in any modern car. By requiring cryptographic authentication from a RFID transponder embedded within the key fob, the immobiliser prevents an attacker from hot-wiring a car. The car typically authenticates the low-frequency transponder through a challenge-response protocol based on symmetric key cryptographic primitives. However, the use of proprietary cryptography in several widespread immobiliser systems has rendered these practically ineffective [22, 215, 216, 222, 89]. In order to safeguard the security of the immobiliser, it is crucial that the community can publicly scrutinise the underlying cryptographic primitives.

Texas Instruments (TI) produced their first cryptographically enabled transponder in 1995, named Digital Signal Transponder 40 (DST40) [98]. Bono et al. reverse engineered the DST40 cipher using a black-box approach in 2005 and showed that an attacker could exhaustively search the key space using an FPGA cluster [22]. As a consequence, TI released DST80 as a successor to the insecure DST40 in 2008 in order to prevent exhaustive search attacks. DST80 transponders use an 80-bit cryptographic key to encrypt a 40-bit challenge generated by the reader. To this date, DST80 remains confidential and the immobiliser systems relying on it (shown in Table 6.1) have not been publicly scrutinised.

**Impact and responsible disclosure.** Based on table 6.1, we assume most Toyota vehicles rolled out between 2010 and 2014, and most Kia/Hyundai vehicles since 2010 are vulnerable to the attacks described in this chapter. Thus, with an average of ~8 million Toyota vehicles produces per year in that period, we estimate ~40 million Toyota vehicles are affected. Likewise, based on a 10-year period, roughly 20 million Kia/Hyundai vehicles are affected [91].

We informed Toyota, Kia, Hyundai and Tesla of the identified issues and provided each with a tailored report. Additionally, we informed Texas Instruments about our intention of publishing the DST80 cipher and provided them with details on the downgrade and

| Make | Period | Model | Make | Period | Model |
|---|---|---|---|---|---|
| Toyota | 2009-2013 | **Auris (2011)** | Kia | 2012+ | **Ceed (2016)** |
| | 2010-2013 | Camry | | 2014 | **Carens (2014)** |
| | 2010-2014 | Corolla | | 2011-2017 | Rio |
| | 2011-2016 | FJ Cruiser | | 2013+ | Soul |
| | 2009-2015 | Fortuner | | 2013-2015 | Optima |
| | 2010+ | Hiace | | 2011+ | Picanto |
| | 2008-2013 | Highlander | Hyundai | 2008+ | I10 |
| | 2009-2015 | **Hilux (2014)** | | 2009+ | I20 |
| | 2009-2015 | Land Cruiser | | 2009+ | I20 |
| | 2011-2012 | RAV4 | | 2010+ | Veloster |
| | 2010-2014 | Urban Cruiser | | 2016 | **IX20 (2016)** |
| | 2011-2013 | Yaris | | 2013 | **I40 (2013)** |
| Tesla | 06/2018-07/2019[1] | **Model S (2018)** | | | |

[1] Tesla resolved the issue using an OTA update allowing affected customers to self-service their key fob.

Table 6.1: Non-exhaustive list of vehicles affected by the research in this chapter. The indicated production period is based on the information available in [31, 207]. The models in bold point out the specific vehicles we inspected.

side-channel attacks. Per request of the manufacturers we redacted the constants in Algorithms 4 and 5.

## 6.2 Contributions

The contributions of this chapter can be summarised as follows:

- **Recovering and reverse engineering immobiliser firmware.** We present a novel fault attack to recover the firmware of the Renesas 78K0 microcontroller, reducing the complexity of the so far most efficient known attack introduced in [25] from 15,000 to two successful power glitches. We reverse engineered the proprietary DST80 cipher from this immobiliser firmware, and we publicly document this cipher for the first time in full detail. Additionally, we recovered the key diversification schemes from three major manufacturers.
- **Security analysis of key diversification schemes.** We analyse the security of the key diversification schemes used in Toyota, Kia and Hyundai immobiliser

systems. We reduce the complexity for recovering the cryptographic key from $2^{80}$ to $2^{24}$ off-line encryptions for Kia and Hyundai. For Toyota immobiliser systems, we show how the key is based on publicly readable information such as the transponder serial number, therefore losing all of its entropy.

- **Practical attacks on DST80 transponders.** We show how a downgrade attack can reduce the key space from $2^{80}$ to $2^{41}$. Depending on the exact configuration, this can lead to recovery of the DST80 key using two lookup tables and only four challenge response pairs. This attack affects the second version of the Tesla Model S key fob. Moreover, we describe Denial-Of-Service attacks, which can render the key fob unusable.

## 6.3 The DST80 Cipher

We present the full details of DST80 here. Unlike the black-box approach taken to recover DST40 in [22], we reverse engineered the DST80 cipher from immobiliser firmware.

### 6.3.1 Reading out immobiliser firmware

We encountered the same immobiliser ECU in both Kia and Hyundai cars. An 8-bit microcontroller (the STM8AF6266 [193]) controls the ignition coil and enables or disables the Engine Control Unit over a serial connection depending on whether the authentication was successful. This particular chip is part of the STM8AF series and has 8kB of flash and an internal EEPROM of 384 bytes. It also features memory read-out protection, however, in this case, the protection was not enabled. Thus, we could recover the firmware and internal EEPROM by connecting to the SWIM interface and issuing a Read-On-The-Fly (ROTF) command, which allows to read any byte in the 24-bit address space. If the read-out protection was enabled, an attacker could resort to the attack described in Chapter 5.

On the other hand, in Toyota cars, a dedicated ECU produced by Tokai Rika handles the immobilisation process. It contains an external ST96320 EEPROM chip connected to a main MCU with only a Tokai Rika part number on it, making it hard to identify the chip. In order to read out the firmware of this MCU, we had to determine its model and debug interfaces. By looking at the locations of standard pins, such as $V_{cc}$ and $V_{ss}$, we narrowed down the search to the Renesas 78K0 series. By taking the location of the oscillator pins (`X1` and `X2`) into account, the 78K0/Kx2 resulted as the most likely candidate. The 78K0 microcontrollers contain an on-chip bootloader, which is accessible through the *Renesas Flash Programming Interface* [166]. By setting the FLMD0 pin high on reset, an external programmer can access this serial interface to write, erase, and verify the internal flash. However, the 78K0 series does not provide a command to read the memory. We used a Raspberry Pi 3 to communicate with the chip using a combination of GPIO pins (for RESET and FLMD0 signals) and the UART pins. After issuing the *signature* command, we recovered the exact model of the MCU, namely the *μPD78F0515A*, a 64kB A-grade 78K0/KC2 MCU [167].

Bozzato et al. show several ways to read out the internal flash of such MCUs through voltage glitching attacks in [25]. One attack is based on overwriting an empty flash section of the MCU with a custom piece of code that outputs the memory over a serial interface, and finally overwriting the reset vector to boot from this section. We improved on this attack by targeting the *Security Set* command. The 78K0/Kx2 series provides several levels of security: *block erase*, *chip erase*, *write* and *boot sector overwrite* protection, which cannot be reversed once enabled. The *Security Set* command sends a byte containing each security bit individually, after which the Renesas bootloader checks that none of the given bits reverse the current security settings. By voltage glitching this command, we disabled all security bits on the MCU, giving us the ability to erase and overwrite the boot section with our custom program to dump the firmware over a serial interface. This attack only requires a single successful glitch. To read the boot section as well, we repeated the attack with a second, identical immobiliser ECU, however now overwriting a different, reachable section with our own program. Once the microcontroller executes this section, the boot section is dumped and we have recovered the whole firmware. Because we need to perform the attack twice in this case, we require a total of two successful glitches, improving significantly on the so far most efficient known attack in [25], which required a total of 15,000 successful glitches.

**Glitch parameters.** In order to read out the immobiliser ECU firmware, we implemented large parts of the Renesas Flash Programming Interface in Python. We acquired a 2011 Toyota Auris and a 2014 Toyota Hilux immobiliser ECU. We desoldered the chips from the ECUs, placed each of them on a breakout board, and connected a 16 MHz resonator to the oscillator pins `X1` and `X2`. We connect the voltage output from the GIAnT to the *REGC* pin in order to bypass the internal regulatory capacitor of the 78K0. Next. we confirmed that both chips contain the same firmware by comparing the values obtained by the *checksum* command.

With a glitch voltage of 0 V and a normal operating voltage of 2.7 V, we triggered on the first transmitted bit of the *Security Set* message from the Raspberry Pi, resulting in a successful glitch of 100 ns width at an offset of 596.78 $\mu$s on the first and 818.05 $\mu$s on the second chip. We attribute the significant difference in glitch offsets to the initial value of the security byte: in the Hilux MCU, the write protection for non-boot sectors was already disabled, while in the chip desoldered from the Auris immobiliser, all protection bits were initially set.

### 6.3.2 Reverse engineering the cipher

Due to most of the DST80 transponders being backwards compatible with DST40, our first assumption was that large parts of the cipher would be quite similar. Therefore, we statically analysed the firmware searching for the well known DST40 Feistel network. We located the equivalent of the $f$-boxes, $g$-boxes and $h$-box as used in DST40 in the firmware, and from there we could start reverse engineering the internals of the cipher. Due to the atypical number of rounds (200), we managed to locate the round function and more importantly, the round key schedule.

### 6.3.3 Cipher details

The DST80 cipher is, like DST40, an unbalanced Feistel network which runs for 200 rounds to calculate the response. It uses an 80-bit cryptographic key, split over two independent Key State Registers. The 40-bit internal state is initialised with the 40-bit challenge generated by the reader and updated with 2 bits each round.

**Definition 6.** Let $b_7 \ldots b_0$ be the bit representation of byte $b$. Then the permutation $P_1(b) : \mathbb{F}_2^8 \to \mathbb{F}_2^8$ is defined as follows.

$$P_1(b_7 \ldots b_0) = \begin{pmatrix} b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \\ b_7 & b_3 & b_5 & b_1 & b_6 & b_2 & b_4 & b_0 \end{pmatrix}$$

**Definition 7.** Let $(S_{n-1}, S_{n-2} \ldots S_1, S_0)$ be the byte representation of $S$. Given Permutation $P_1$ as defined in Definition 6, then we define the operation $P_2(S)$ as follows.

$$P_2(S) \leftarrow (P_1(S_{n-1}), P_1(S_{n-2}), \ldots, P_1(S_1), P_1(S_0))$$

At the start of each round as depicted in Algorithm 3, the function `DST80_Merge(l, r)` combines the two 40-bit key state registers into one 40-bit round key as shown in Algorithm 2. It permutes the bytes of both keys according to $P_1$, after which it inverts each register depending on the value of its most significant bit. Finally, it returns a 40-bit round key which serves along with the internal state as input to the Feistel function $F$. This Feistel function is identical to its DST40 equivalent, resulting in a pair of bits which is fed back into the challenge register. For reference, we included the specification of the Feistel function in Appendix C. Unlike DST40 (where the key state register is shifted every three rounds) the LFSRs containing the two 40-bit keys are shifted each round, with tapped bits on positions 0, 2, 19 and 21.

**Algorithm 2** Generation of the DST80 round key

```
function DST80_Merge(keyL, keyR)
    keyL ← $P_2$(keyL)
    keyR ← $P_2$(keyR)
    if $\texttt{keyL}_{39} == 1$ then
        keyL ← keyL ⊕ 0x7FFFFFFFFF
    end if
    if $\texttt{keyR}_{39} == 1$ then
        keyR ← keyR ⊕ 0x7FFFFFFFFF
    end if
    return $(\texttt{keyL}_{39...20}, \texttt{keyR}_{39...20})$
end function
```

**Algorithm 3** The DST80 round function

```
function DST80_Round(keyL, keyR, $s$)                    ▷ With $s$ = internal state
    $k \leftarrow DST80\_Merge(\texttt{keyL}, \texttt{keyR})$
    $s \leftarrow (s \gg 2) \mid ((F(k, s) \oplus s_{1...0}) \ll 38)$
    $\texttt{keyR} \leftarrow (\texttt{keyR} \gg 1) \mid ((\texttt{keyR}_0 \oplus \texttt{keyR}_2 \oplus \texttt{keyR}_{19} \oplus \texttt{keyR}_{21}) \ll 39)$
    $\texttt{keyL} \leftarrow (\texttt{keyL} \gg 1) \mid ((\texttt{keyL}_0 \oplus \texttt{keyL}_2 \oplus \texttt{keyL}_{19} \oplus \texttt{keyL}_{21}) \ll 39)$
end function
```

## 6.4 Practical attacks on DST80 systems

This section describes several practical attacks on real-world DST80 systems. Through reverse engineering the immobiliser firmware, we present and analyse the security of two key diversification schemes used by three major car manufacturers. Furthermore, we illustrate how configuration issues of a transponder can lead to Denial-of-Service and downgrade attacks.

### 6.4.1 Uncovering key diversification schemes from immobiliser firmware

The security of the DST80 cipher relies on 80 bits of entropy in the cryptographic key, split up in two 40-bit chunks: `keyL` and `keyR`. Ideally, either the car manufacturer or OEM generates a random and unique 80-bit secret key for each paired transponder. However, in practice we have discovered that the key generation process is not random at all. We recovered the secret keys from the immobiliser ECUs of three different manufacturers and find recurring patterns among keys of each of them. In this Section, we bring to light the key diversification schemes we have recovered from studying the immobiliser firmware. With knowledge of this key generation scheme hidden in the immobiliser firmware an attacker can recover cryptographic keys from a DST80 transponder with only a single challenge response pair.

**Attacker model.** For the attacks described here, we assume that the adversary can communicate with the DST80 transponder, either wirelessly or over a serial interface.

#### Kia and Hyundai low-entropy keys

While the microcontroller's flash memory is typically quite large and thus used for program code, the EEPROM usually contains small amounts of data, specific to the car or even ECU. After having located serial numbers of the paired transponders in the EEPROM, we deemed it very likely that the corresponding DST80 keys would be stored there as well. Since the internal EEPROM of the microcontroller on the immobiliser is only 384 bytes, it is plausible to recover the DST80 key by doing an exhaustive search on the EEPROM data. First, we generate a challenge and acquire the corresponding signature from the

| **Make** | **Year & Model** | `keyL` | `keyR` |
|---|---|---|---|
| Kia | 2014 Carens | `955A3E,X` | `Y,C1A56A` |
| | 2016 Ceed | `724560,X` | `Y,9FBA8D` |
| Hyundai | 2013 I40 | `756F1E,X` | `Y,E1908A` |
| | 2016 IX20 | `357B13,X` | `Y,EC84CA` |

Table 6.2: Kia and Hyundai immobilisers and their respective DST80 keys. The two 2-byte constants `X` and `Y` have been redacted on the manufacturers request.

transponder. Assuming the `keyL` and `keyR` are adjacent in memory, we set each sequence of 10 bytes as the cryptographic key and check whether they produce the same response. In order to eliminate any problems caused by endianness, we compute a signature with every key byte at every position in the 10-byte key. Note that for larger EEPROMs we can perform an entropy analysis to identify the areas most likely to contain a cryptographic key. Furthermore, if the preceding procedure does not produce a candidate key, we can enhance it by searching the binary in blocks of five bytes, or in the worst case we can search by byte. This can however result in state explosion and becomes infeasible for larger EEPROM sizes.

Using these techniques, we recovered the ten key bytes displayed in Table 6.2 from the internal EEPROM. A first observation is that even though two different physical transponders are paired to the immobiliser, they share the same cryptographic key. We can confirm this without knowledge of the DST80 key by sending the same challenge to both transponders, which generate identical responses.

Furthermore, Table 6.2 suggests that two bytes of both the `keyL` and the `keyR` are identical in all four of the DST80 keys, leaving six unknown bytes. Algorithm 4 clarifies the key generation scheme used by Kia and Hyundai, leaving only three bytes of entropy. Thus, provided they have brief access to a legitimate key fob, an attacker can immediately recover a valid DST80 key from a Kia/Hyundai transponder with only one challenge-response pair.

**Algorithm 4** Recovering DST80 keys for transponders configured for Hyundai and Kia vehicles. The two 2-byte constants $X$ and $Y$ have been redacted on the manufacturers request.

```
function SEARCH_KEY(C, S)                          ▷ With C - Challenge; S - Signature
    for $i$ in $\{0 \dots 2^{24}\}$ do
        $key_L[4], key_L[3], key_L[2] \leftarrow i[0], i[1], i[2]$
        $key_L[1], key_L[0] \leftarrow \mathtt{X}$
        $key_R[4], key_R[3] \leftarrow \mathtt{Y}$
        $key_R[2] \leftarrow \neg key_L[2]$
        $key_R[1] \leftarrow \neg key_L[3]$
        $key_R[0] \leftarrow \neg key_L[4]$
        $crc, sig \leftarrow DST80(C, key_L, key_R)$
        if $sig == S$ then return $key_L, key_R$
        end if
    end for
end function
```

**Attack complexity.** This attack only requires one challenge-response pair. We reduced the computational complexity to recover a cryptographic key from a Kia/Hyundai DST80 transponder from $2^{80}$ to $2^{24}$ encryptions.

### Toyota key diversification scheme

We reverse engineered the Toyota immobiliser firmware to recover and analyse the security critical procedures contained within. Algorithm 5 depicts one of these procedures we reverse engineered from the firmware, namely the key derivation scheme for the DST80 cryptographic keys. Every time a transponder is presented, the immobiliser derives the DST80 key from its serial number (stored on page 3) and the 1-byte values stored on pages 1 and 2. Since the least significant byte of the DST80 transponder ID is the manufacturer code, this byte does not affect the key generation. Besides these three always readable transponder pages, the key derivation scheme relies on three security constants stored in the internal flash memory of the immobiliser. From our experiments, these security constants are identical across all Toyota DST80 immobilisers analysed. The fact that they are located in flash memory strengthens this hypothesis, since flash memory is less commonly used to store unique secrets.

**Algorithm 5** The Toyota key generation algorithm reverse engineered from immobiliser firmware. The three 5-byte constants $X_0 \dots X_2$ have been redacted on the manufacturers request.

```
$X \leftarrow [X_0, X_1, X_2]$
**function** GEN_KEY(page 1, page 2, id)
    $key_L \leftarrow (id \ll 16) \mid (page1 \ll 8) \mid page2$
    $key_R \leftarrow key_L$
    **for** $i$ in $\{0 \dots 7\}$ **do**
        $S_1 \leftarrow key_R$
        $S_2 \leftarrow key_R \oplus X[i \mod 3]$
        $S_3[0] \leftarrow ((S_2[0] + S_2[2] + S_2[3]) \,\&\, \texttt{ff}) \lll 1$
        $S_3[1] \leftarrow ((S_2[2] + S_2[3] + S_2[4]) \,\&\, \texttt{ff}) \lll 3$
        $S_3[2] \leftarrow ((S_2[0] + S_2[1] + S_2[3]) \,\&\, \texttt{ff}) \lll 1$
        $S_3[3] \leftarrow ((S_2[0] + S_2[1] + S_2[4]) \,\&\, \texttt{ff}) \lll 3$
        $S_3[4] \leftarrow ((S_2[1] + S_2[2] + S_2[4]) \,\&\, \texttt{ff}) \lll 1$
        $key_R = S_3 \oplus key_L$
        $key_L = S_1$
    **end for**
     **return** $key_L, key_R$
**end function**
```

**Attack complexity.** This attack only requires reading out three *public* transponder pages. Therefore, using the uncovered key derivation scheme, the security of this system is effectively reduced from $2^{80}$ to a few operations.

## 6.5 Transponder configuration issues

In Section 2.4.2 we outlined the different DST transponders available. The main difference in these transponders lies in the available interfaces. The wedge type transponder has a Low Frequency (LF) interface whereas the TMS37126 has both a Serial Peripheral Interface (SPI) and an LF interface. The TMS37F128 has both interfaces but the SPI is bonded internally to the MSP430 making it more difficult to access for an adversary.

These interfaces can be used to configure the transponder, to read and write values from and to the EEPROM storage and to request a cryptographic response to a provided challenge. All accesses to these interfaces are in fact accesses to an EEPROM page, each of which is five bytes in size and can be locked to prevent modification. For example, a cryptographic key can be set by writing to page 4 of the transponder. Afterwards, this page can be locked to prevent modification. By default a new DST transponder is configured to compute DST40 responses to a provided challenge. During car manufacturing or key fob pairing the transponder can be configured to use “80-bit mode” (DST80) by setting the correct bit in page 30 of the transponder.

In this section we will discuss multiple vulnerabilities which can be exploited using these interfaces when the transponder is not correctly configured during key fob manufacturing or pairing. We will assume that the transponder was configured to use DST80 and will discuss how configuration issues can reduce the security level provided by these transponders.

### Downgrade attacks

We identified three distinct scenarios in which a downgrade attack could be used to reduce the security provided by these transponders. The first two scenarios do not require physical access to the key fob whereas the third scenario does require physical access.

**Page 30 unlocked.** A transponder configured to use DST80 with page 30 left unlocked can be downgraded to use DST40 instead. In this scenario, the transponder uses `keyL`, one of the two 40-bit keys used in DST80, to compute a DST40 response. This means that an adversary can downgrade the transponder and recover half of the 80-bit key using only two challenge response pairs, effectively reducing the computational complexity of the attack from $2^{80}$ to $2^{41}$ encryptions. After reverting back to 80-bit mode the transponder will again use the original 80-bit key. This attack requires short-range LF communication.

**Page 4 and page 30 unlocked.** Recent work shows that a DST40 key can be recovered using a 5.4 TB precomputed table, the response to a chosen challenge, a second challenge response pair and $2^{15}$ DST40 operations on average [222]. Our experiments show that we can recover a DST80 key with only twice the amount of resources if, in addition to page 30, page 4 is left unlocked. Similar to the previous scenario this attack requires short-range LF communication.

Specifically, an adversary can first alter the transponder's configuration to use DST40 instead of DST80 (because page 30 is left unlocked). The transponder will now use `keyL` with the DST40 cipher to compute the response to a provided challenge. At this point the adversary can use the attack as described in [222] to recover `keyL`. After recovering the 40-bit key we can change it to a chosen value (because page 4 was left unlocked) and we can reconfigure the transponder to use DST80. Empirical results show that in this scenario the transponder will compute DST80 responses using the original `keyR` in combination with the chosen `keyL`. The last step is now to recover `keyR` which can be achieved using a second precomputed table. We can generate this second table by computing the responses to a chosen 40-bit challenge for all possible values of `keyR` while keeping `keyL` fixed to a chosen value. In fact, we can do this more efficiently if we choose `keyL` to be the all 0 key. In that case the contents of the LFSR will remain constant throughout the 200-round execution of DST80 and most of the $f$-functions can be simplified because of a constant input, resulting in a more efficient software implementation.

We discovered that the Tesla Model S key fob released as a response to the attacks shown in [222] was not properly configured. These key fobs had both page 4 and page 30 unlocked, allowing for key recovery with approximately twice the effort compared to the previous attack. The downgrade attack reduces the computational complexity from $2^{80}$ to $2^{41}$ encryptions. Additionally, we only require 4 challenge response pairs and two 5.4 TB precomputed tables to recover the full 80-bit key in a matter of seconds.

Tesla was able to resolve the issue using an Over-The-Air software update that allowed customers to self-service their key fobs.

**Page 4 and page 30 locked.** The final scenario requires physical access to the key fob and arises when both page 4 and page 30 are locked down by the manufacturer. If the target chip has an easily accessible SPI interface the adversary can control the transponder from this interface and request DST40 responses even if the transponder is configured to use DST80, again reducing the computational complexity of the attack from $2^{80}$ to $2^{41}$. While this attack is the easiest to apply in the case of a TMS37126 transponder it can also be applied to the TMS37F128 by exposing the bond wires interconnecting the MSP430 and TMS37126 or by replacing the firmware image running on the MSP430.

If the MSP430 does not have its JTAG fuse blown, a Commercial-Off-The-Shelf (COTS) MSP430 programmer can be used to create a backup of the firmware before replacing it with a malicious version. The malicious firmware image can query the transponder for the required challenge response pairs after which the adversary can restore the original firmware image. If the JTAG fuse is blown the adversary will first have to obtain a legitimate firmware image from a single key fob. They could achieve this by exploiting a vulnerability in the MSP430 bootloader such as described in Chapter 5. Once the adversary has obtained a copy of the firmware, they can use the Interrupt Vector Table (IVT) stored at the end of flash as a password to unlock the bootloader in subsequent attacks. The remainder of the attack is the same as before; replace the firmware to get challenge response pairs and rewrite the original firmware image using the MSP430 bootloader instead of the JTAG interface.

### Denial-of-Service attacks

Leaving transponder pages unlocked can lead to trivial Denial-of-Service (DoS) attacks. For example, if the configuration (page 30) is not write protected an adversary could change the transponder to use DST40 instead of DST80 (or vice versa) to prevent a user from starting their vehicle. Similarly, if the cryptographic key is not write protected an adversary can overwrite it to prevent a user from starting their vehicle. If their only goal is DoS they can additionally lock these pages permanently, preventing a dealership from repairing the key fob.

Performing this type of DoS attack can be automated by building a device which repeatedly broadcasts the required commands. While there might be little incentive for someone to do this type of attack it could lead to bad publicity for the affected car manufacturers and increased revenue for local garage owners.

## 6.6 Discussion and mitigation

With regard to the weak key generation based attacks described in Section 6.4.1, we believe that one reason why manufacturers choose to implement such a scheme is to facilitate the key pairing process. For instance, certain Toyota models allow programming a new key fob by inserting it into the ignition and performing a precise sequence of actions [206]. This is only possible because the immobiliser can derive the DST80 key from transponder pages 1-3 as described in Section 6.4.1. Instead of relying on a weak key generation scheme, the manufacturer can mitigate this by implementing a secure-diagnostics based solution for pairing a new key, such as the one we propose in Chapter 4.

It is of the utmost importance that DST transponders are configured correctly to offer any security guarantees. Any car manufacturer or OEM using these transponders should ensure that page 4 and page 30 are locked. Furthermore, locking other transponder pages should be considered if they are not used. While migrating the key programming procedure is an involved task for the manufacturer, the misconfigured transponders described in Section 6.5 are easier to fix. For instance, Tesla fixed the issues with their DST80 key fobs with a software update issued a few months after disclosure. Finally, enabling mutual authentication on the transponder would mitigate transponder-only attacks as described in Sections 6.4.1 and 6.5. These require a challenge response pair, which an attacker could not obtain without knowing the encryption key. However, if the adversary has access to the car they can still acquire a challenge response pair by eavesdropping the wireless communication between the transponder and immobiliser ECU.

**Security of DST80** We would like to emphasise that by no means we make any claims about the cryptographic security of the DST80 cipher here. The attacks described in this chapter are entirely practical of nature and target specific transponder key diversification and configuration issues in immobiliser systems by certain manufacturers. Regarding the cryptographic security of the cipher, the use of an 80 bit secret key addresses the main issue of its predecessor DST40, which was broken with an exhaustive search of the 40 bit key space in [22]. However, we hope that by reverse engineering and publishing the cipher here, the community can publicly scrutinize it, and either refute or prove its security.

## 6.7 Chapter Summary

In this chapter we demonstrated the insecurity of two different immobiliser systems used by three major car manufacturers. By glitching the firmware protection of the microcontroller we read the firmware of several immobiliser ECUs. From this firmware, we reverse engineered the proprietary Texas Instruments DST80 cipher. We present the cipher here for the first time, which enables other researchers to scrutinise its security. Additionally, we demonstrate that Toyota derives an 80-bit cryptographic key for each transponder based on its serial number and other publicly readable transponder data. Similarly, we show how transponders used in Kia and Hyundai cars rely on three bytes of entropy in the encryption key. In each of these cases an attacker can recover the full cryptographic key in a matter of milliseconds. Additionally, we show how insufficient diligence from the manufacturer can lead to a vulnerable configuration of the transponder, resulting in practical key recovery and DoS attacks.

This wide range of attacks summarised in Table 6.3 demonstrates the large attack surface of security critical systems like the vehicle immobiliser. An effective immobiliser system does not only require secure cryptographic primitives, but also a secure key generation scheme, rigorously configured transponders and secure hardware implementations.

| **Affected** | **Prerequisites** | **Consequences** |
|---|---|---|
| Toyota<br>Kia<br>Hyundai | Communication with the transponder<br>Single authentication trace | Key recovery<br>(Algorithms 4 and 5) |
| Tesla | Write access to transponder pages 4 and 30 | complexity $2^{41}$* |
| | Write access to transponder page 30 | complexity $2^{41}$* |

Table 6.3: Attacks on DST80 transponders described in this chapter. In the scenarios marked by an asterisk lookup tables can be employed to speed up the key recovery process.

In this chapter we address each of these elements and describe several countermeasures to mitigate the proposed attacks.

# Part III

# Closing Statements

# CHAPTER 7

# CLOSING REMARKS

In this chapter we draw conclusions from the work presented in this thesis.

## 7.1 Conclusion

The automotive industry has made big technological advances in the past decades. Unfortunately, this has also come at a cost of diminished security, leading to an environment comparable to commodity software in the nineties. This is once again evidenced by the work presented in this thesis.

Firstly, we address *why* automotive firmware analysis is crucial for a more secure automotive environment. We uncover proprietary cryptography and low entropy internal cipher states used in the authentication mechanism for diagnostic protocols. This exposes critical functionality, such as read and write access to memory, to an attacker connected to the internal network. Furthermore, we reverse engineer and present DST80, a proprietary immobiliser cipher, and propose several attacks on DST80-based immobiliser systems by exploiting weak key generation schemes and transponder configuration issues. The use and/or insecure deployment of these proprietary ciphers once again underlines the importance of public scrutiny of automotive components. However, to achieve that the analyst must be able to recover the firmware from the ECU under test, which is often read protected.

Next, we direct our efforts at *what* to analyse. To that purpose, we propose several firmware extraction techniques for microcontrollers embedded on ECUs. On the one hand, we present a firmware extraction framework which bypasses diagnostic authentication and solely uses diagnostic functionality present on the ECU to download and execute code in its memory. On the other hand, we combine low-level software and hardware exploits with dynamic and static firmware analysis techniques to exploit readout protection mechanisms on embedded bootloaders.

Finally, we address *how* to analyse automotive systems. To that end, we perform an end-to-end security analysis of several real-world immobiliser systems. Furthermore, we enhance existing firmware analysis and exploitation techniques and port these to the realm of low-level automotive firmware.

To conclude, the work presented in this thesis brings a much needed step in the direction of a more open automotive ecosystem, ultimately leading the path to bring the state of the art in commodity software and embedded security to these esoteric and intricate automotive systems.

# APPENDIX A

# FULL ASSEMBLY OF THE `GET_BLOCK_NO` AND `CMP_ADDR` FUNCTIONS FROM THE 78K0 BOOTLOADER

Listing A.1: Assembly of the `get_block_no` function in the 78K0 bootloader. Registers `a`, `b` and `c` are depicted in lower case, while the input address $A = A_2A_1A_0$ (with each address byte referencing a location in RAM) is given in uppercase

```
get_block_no:   push    bc
                cmp     A0, #0FFh
                bz      _b3
                cmp     A1, #00h
                bz      _b4
                mov     a, A0
                xch     a, X
                mov     a, A1
                br      calc_blk
_b4:
                mov     a, #00h
                br      _b6
_b3:
                cmp     A1, #0FFh
                bnz     enter_calc_blk
                mov     a, #3Fh
                br      _b6
enter_calc_blk:
                mov     a, A_0
                add     a, #01h
                xch     a, X
                mov     a, A_1
                addc    a, #00h
calc_blk:
                mov     C, #08h
                divuw   C
                mov     C, #80h
                divuw   C
                xch     a, X
                cmp     A_0, #0FFh
                bnz     _b6
                dec     a
_b6:
                cmp     A_2, #01h
                bnz     return
                add     a, #40h

return:
                pop     bc
                ret
```

Listing A.2: Assembly of the `cmp_addr` function in the 78K0 bootloader with input addresses $A = A_2A_1A_0$ and $B = B_2B_1B_0$.

```
cmp_addr:         mov     a, A2
                  cmp     a, B2
                  bc      ret_0
                  bnz     loc_874
                  mov     a, A1
                  cmp     a, B1
                  bc      ret_0
                  bnz     loc_874
                  mov     a, A0
                  cmp     a, B0
                  bc      ret_0
                  bnz     loc_874
ret_0:
                  clr1    CY
                  ret
ret_1:
                  set1    CY
                  ret
```

# APPENDIX B

# EXAMPLE PATH THROUGH THE `CHECKSUM` COMMAND HANDLER

Listing B.1: Example path through the checksum command handler for equivalence class `fc`. Our technique marks input bytes (`[HL + 00]` ,. . ., `[HL + 05]`) as symbolic and builds up the constraints along the path.

```
addr  Instruction                                      cycles


1aa8   call !sanity_check_addr                            7
 892   set1 flash_getbyte_reg.03h                         4
 895   call !sub_FE9                                      7
 fe9   mov A, #0FFh                                       4
 feb   call !sub_103F                                     7
103f   mov !d_resp_0, A                                   8
1042   mov A, #01h                                        4
1044   mov !byte_FE14, A                                  8
1047   ret                                                6
 fee   ret                                                6
 898   movw HL, #msg_buffer_b1                            8
 89b   mov A, [HL+03h]                                    8
 89d   mov addr_H, A                                      4
 89f    mov end_addr_H, A          4
 8a1    mov A, [HL+04h]            8
 8a3    mov addr_M, A              4
 8a5    mov end_addr_M, A          4
 8a7    mov A, [HL+05h]            8
 8a9    mov addr_L, A              4
 8ab    mov end_addr_L, A          4
 8ad    call !get_block_no         7
117d    push BC                    4
117e    cmp addr_L, #0FFh          6
1181    bz loc_1193                6
1193    cmp addr_M, #0FFh          6
1196    bnz loc_119C               6
119c    mov A, addr_L              4
119e    add A, #01h                4
11a0    xch A, X                   2
11a1    mov A, addr_M              4
11a3    addc A, #00h               4
11a5    mov C, #08h                4
11a7    divuw C                    25
11a9    mov C, #80h                4
11ab    divuw C                    25
11ad    xch A, X                   2
11ae    cmp addr_L, #0FFh          6
11b1    bnz loc_11B4               6
11b3    dec A                      2
11b4    cmp addr_H, #01h           6
11b7    bnz loc_11BB               6
```

```
11bb    pop BC                          4
11bc    ret                             6
 8b0    mov end_block_no_0, A           4
 8b2    mov A, [HL+00h]                 8
 8b4    mov addr_H, A                   4
 8b6    mov A, [HL+01h]                 8
 8b8    mov addr_M, A                   4
 8ba    mov A, [HL+02h]                 8
 8bc    mov addr_L, A                   4
 8be    call !get_block_no              7
117d    push BC                         4
117e    cmp addr_L, #0FFh               6
1181    bz loc_1193                     6
1183    cmp addr_M, #00h                6
1186    bz loc_118F                     6
118f    mov A, #00h                     4
1191    br loc_11B4                     6
11b4    cmp addr_H, #01h                6
11b7    bnz loc_11BB                    6
11bb    pop BC                          4
11bc    ret                             6
 8c1    mov start_block_no, A           4
 8c3    mov A, end_block_no_0           4
 8c5    sub A, start_block_no           4
 8c7    inc A                           2
 8c8    mov diff_block_no, A            4
 8ca    call !comp_end_addr             7
 85a    mov A, end_addr_H               4
```

| | | |
|---|---|---|
| 85c | **cmp** A, max_flash_addr_H | 6 |
| 85e | **bc** loc_872 | 6 |
| 860 | **bnz** loc_874 | 6 |
| 862 | **mov** A, end_addr_M | 4 |
| 864 | **cmp** A, max_flash_addr_M | 6 |
| 866 | **bc** loc_872 | 6 |
| 872 | **clr1** CY | 2 |
| 873 | **ret** | 6 |
| 8cd | **bc** loc_874 | 6 |
| 8cf | **mov** A, addr_H | 4 |
| 8d1 | **cmp** A, end_addr_H | 6 |
| 8d3 | **bc** loc_8E7 | 6 |
| 8d5 | **bnz** loc_8E5 | 6 |
| 8d7 | **mov** A, addr_M | 4 |
| 8d9 | **cmp** A, end_addr_M | 6 |
| 8db | **bc** loc_8E7 | 6 |
| 8dd | **bnz** loc_8E5 | 6 |
| 8df | **mov** A, addr_L | 4 |
| 8e1 | **cmp** A, end_addr_L | 6 |
| 8e3 | **bc** loc_8E7 | 6 |
| 8e7 | **clr1** CY | 2 |
| 8e8 | **ret** | 6 |
| 1aab | **bc** error_5 | 6 |
| 1aad | **cmp** addr_L, #00h | 6 |
| 1ab0 | **bnz** error_5 | 6 |

# APPENDIX C

# SPECIFICATION OF THE DST80 FEISTEL FUNCTION

We specify the Feistel function $F(k, s)$, originally defined in [22], again. We present it here as we reverse engineered it from the immobiliser firmware.

$$F(k, s) = h(g_4, g_3, g_2, g_1)$$

$$\begin{aligned}
g_1 &= g(f_{16}, f_{15}, f_{14}, f_{13}) \\
g_2 &= g(f_{12}, f_{11}, f_{10}, f_9) \\
g_3 &= g(f_8, f_7, f_6, f_5) \\
g_4 &= g(f_4, f_3, f_2, f_1)
\end{aligned}$$

$$\begin{aligned}
f_1 &= f_\delta(s_{32}, k_{32}, s_{24}, k_{24}, s_{16}) \\
f_2 &= f_\epsilon(k_{16}, s_8, k_8, k_0) \\
f_3 &= f_\beta(s_{33}, k_{33}, s_{25}, k_{25}, s_{17}) \\
f_4 &= f_\epsilon(k_{17}, s_9, k_9, k_1)
\end{aligned}$$

$$f_5 = f_\delta(s_{34}, k_{34}, s_{26}, k_{26}, s_{18})$$

$$\begin{aligned}
f_6 &= f_\gamma(k_{18}, s_{10}, k_{10}, s_2, k_2) \\
f_7 &= f_\beta(s_{35}, k_{35}, s_{27}, k_{27}, s_{19}) \\
f_8 &= f_\alpha(k_{19}, s_{11}, k_{11}, s_3, k_3)
\end{aligned}$$

$$\begin{aligned}
f_9 &= f_\delta(s_{36}, k_{36}, s_{28}, k_{28}, s_{20})) \\
f_{10} &= f_\gamma(k_{20}, s_{12}, k_{12}, s_4, k_4) \\
f_{11} &= f_\beta(s_{37}, k_{37}, s_{29}, k_{29}, s_{21}) \\
f_{12} &= f_\alpha(k_{21}, s_{13}, k_{13}, s_5, k_5)
\end{aligned}$$

$$\begin{aligned}
f_{13} &= f_\delta(s_{38}, k_{38}, s_{30}, k_{30}, s_{22}) \\
f_{14} &= f_\gamma(k_{22}, s_{14}, k_{14}, s_6, k_6) \\
f_{15} &= f_\beta(s_{39}, k_{39}, s_{31}, k_{31}, s_{23}) \\
f_{16} &= f_\alpha(k_{23}, s_{15}, k_{15}, s_7, k_7)
\end{aligned}$$

| index | $f_\alpha$ | $f_\beta$ | $f_\gamma$ | $f_\delta$ | $f_\epsilon$ | $g$ | $h$ |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 0 | 0 | 1 | 0 | 3 |
| 3 | 0 | 1 | 0 | 1 | 0 | 0 | 3 |
| 4 | 0 | 0 | 1 | 1 | 0 | 1 | 2 |
| 5 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 6 | 1 | 0 | 1 | 1 | 1 | 1 | 2 |
| 7 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 9 | 0 | 0 | 1 | 1 | 1 | 1 | 2 |
| 10 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 11 | 1 | 0 | 1 | 0 | 0 | 1 | 2 |
| 12 | 0 | 1 | 0 | 1 | 0 | 0 | 3 |
| 13 | 1 | 1 | 0 | 1 | 1 | 0 | 3 |
| 14 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| 15 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 16 | 1 | 1 | 1 | 0 | | | |
| 17 | 0 | 0 | 0 | 0 | | | |
| 18 | 1 | 1 | 1 | 1 | | | |
| 19 | 0 | 0 | 1 | 1 | | | |
| 20 | 1 | 1 | 1 | 0 | | | |
| 21 | 1 | 1 | 0 | 0 | | | |
| 22 | 0 | 0 | 0 | 1 | | | |
| 23 | 0 | 0 | 0 | 1 | | | |
| 24 | 0 | 0 | 0 | 0 | | | |
| 25 | 1 | 0 | 0 | 1 | | | |
| 26 | 0 | 1 | 0 | 0 | | | |
| 27 | 1 | 1 | 1 | 1 | | | |
| 28 | 1 | 0 | 1 | 1 | | | |
| 29 | 1 | 1 | 1 | 0 | | | |
| 30 | 0 | 0 | 0 | 1 | | | |
| 31 | 0 | 1 | 1 | 0 | | | |

Table C.1: The tables used in the Feistel function F

# LIST OF REFERENCES

[1] ISO 11898-1:2015. Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling. Standard, International Organization for Standardization, Geneva, CH, 2015.

[2] ISO 14229-1:2006. Road vehicles — Unified diagnostic services (UDS) — Specification and requirements. Standard, International Organization for Standardization, Geneva, CH, 2006.

[3] ISO 14230-3:1999. Road vehicles — Diagnostic systems — Keyword Protocol 2000 — Part 3: Application layer. Standard, International Organization for Standardization, Geneva, CH, 1999.

[4] ISO 15765-2:2016. Road vehicles — Diagnostic communication over Controller Area Network (DoCAN) — Part 2: Transport protocol and network layer services. Standard, International Organization for Standardization, Geneva, CH, 2016.

[5] Michel Agoyan, Jean-Max Dutertre, David Naccache, Bruno Robisson, and Assia Tria. When Clocks Fail: On Critical Paths and Clock Faults. In Dieter Gollmann, Jean-Louis Lanet, and Julien Iguchi-Cartigny, editors, *Smart Card Research and Advanced Application, 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010, Passau, Germany, April 14-16, 2010. Proceedings*, volume 6035 of *Lecture Notes in Computer Science*, pages 182–193. Springer, 2010.

[6] Ross Anderson and Markus Kuhn. Tamper Resistance - a Cautionary Note. In *Proceedings of the second Usenix Workshop on Electronic Commerce*, volume 2, pages 1–11, 1996.

[7] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *Proceedings of the 25th USENIX Security Symposium*, 2016.

[8] Dennis Andriesse, Asia Slowinska, and Herbert Bos. Compiler-Agnostic Function Detection in Binaries. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy*, 2017.

[9] ARM. CoreSight Trace Memory Controller. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0461b/DDI0461B_tmc_r0p1_trm.pdf`. accessed: 2020/06/29.

[10] ARM. CoreSight Program Flow Trace. `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0035b/IHI0035B_cs_pft_v1_1_architecture_spec.pdf`, 2011.

[11] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. IJON: Exploring Deep State Spaces via Fuzzing. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, 2020.

[12] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. An In-depth and Blackbox Characterization of the Effects of Clock Glitches on 8-bit MCUs. In Luca Breveglieri, Sylvain Guilley, Israel Koren, David Naccache, and Junko Takahashi, editors, *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2011, Tokyo, Japan, September 29, 2011*, pages 105–114. IEEE Computer Society, 2011.

[13] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[14] Alessandro Barenghi, Guido Bertoni, Emanuele Parrinello, and Gerardo Pelosi. Low Voltage Fault Attacks on the RSA Cryptosystem. In Luca Breveglieri, Israel Koren, David Naccache, Elisabeth Oswald, and Jean-Pierre Seifert, editors, *Sixth International Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2009, Lausanne, Switzerland, 6 September 2009*, pages 23–31. IEEE Computer Society, 2009.

[15] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.

[16] Giampaolo Bella, Pietro Biondi, Gianpiero Costantino, and Ilaria Matteucci. TOUCAN: A proTocol tO secUre Controller Area Network. In *Proceedings of the ACM Workshop on Automotive Cybersecurity, AutoSec@CODASPY 2019, Richardson, TX, USA, March 27, 2019*, pages 3–8, 2019.

[17] Ryad Benadjila, Mathieu Renard, José Lopes-Esteves, and Chaouki Kasmi. One Car, Two Frames: Attacks on Hitag-2 Remote Keyless Entry Systems Revisited. In *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017*, 2017.

[18] Eli Biham and Adi Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In *Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology – CRYPTO'97*, pages 513 – 525, 1997.

[19] Andrey Bogdanov. Linear Slide Attacks on the KeeLoq Block Cipher. In Dingyi Pei, Moti Yung, Dongdai Lin, and Chuankun Wu, editors, *Information Security and Cryptology, Third SKLOIS Conference, Inscrypt 2007, Xining, China, August 31 - September 5, 2007, Revised Selected Papers*, volume 4990 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2007.

[20] Wouter Bokslag. An Assessment of ECM Authentication in Modern Vehicles. Master's thesis, Eindhoven University of Technology, 2017.

[21] Dan Boneh, Richard A. Demillo, and Richard J. Lipton. On the Importance of Checking Computations. In *Proceedings of Eurocrypt'97*, pages 37 – 51, 1997.

[22] Steve Bono, Matthew Green, Adam Stubblefield, Ari Juels, Aviel D Rubin, and Michael Szydlo. Security Analysis of a Cryptographically-Enabled RFID Device. In *Proceedings of the 14th USENIX Security Symposium (USENIX Security 2005)*, pages 1–16. USENIX Association, 2005.

[23] Bosch. CAN Specification. Technical report, Robert Bosch GmbH, 1991.

[24] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. Shaping the Glitch: Optimizing Voltage Fault Injection Attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(2):199–224, 2019.

[25] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. Shaping the Glitch: Optimizing Voltage Fault Injection Attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(2):199–224, 2019.

[26] Kris Brosch. Firmware dumping technique for an ARM Cortex-M0 SoC . `https://blog.includesecurity.com/2015/11/NordicSemi-ARM-SoC-Firmware-dumping-technique.html`.

[27] bunniestudios. Hacking the PIC 18F1320. https://www.bunniestudios.com/blog/?page_id=40.

[28] R. Buttigieg, M. Farrugia, and C. Meli. Security Issues in Controller Area Networks in Automobiles. In *2017 18th International Conference on Sciences and Techniques of Automatic Control and Computer Engineering (STA)*, pages 93–98, 2017.

[29] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224, 2008.

[30] Zhiqiang Cai, Aohui Wang, Wenkai Zhang, M Gruffke, and H Schweppe. 0-days & Mitigations: Roadways to Exploit and Secure Connected BMW Cars. *Black Hat USA*, 2019:39, 2019.

[31] Car Keys Online. https://www.car-keys-online.com.

[32] Rafael Boix Carpi, Stjepan Picek, Lejla Batina, Federico Menarini, Domagoj Jakobovic, and Marin Golub. Glitch It If You Can: Parameter Search Strategies for Successful Fault Injection. In *Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*, pages 236–252, 2013.

[33] Silvio Cesare. Adventures in glitching PIC microcontrollers to defeat firmware copy protection. https://2015.kiwicon.org/the-con/talks/#e203.

[34] Silvio Cesare. Defeating Firmware Copy Protection Using Glitching. https://www.youtube.com/watch?v=UGlm_I-RI-U.

[35] Robert N. Charette. This car runs on code. Online: https://spectrum.ieee.org/transportation/systems/this-car-runs-on-code, 2 2009.

[36] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *20th USENIX Security Symposium (USENIX Security 2011)*. USENIX Association, 2011.

[37] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Proceedings of the 2016 Network and Distributed System Security Symposium*, 2016.

[38] Kyong-Tak Cho and Kang G. Shin. Error Handling of In-vehicle Networks Makes Them Vulnerable. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1044–1055, 2016.

[39] Wonsuk Choi, Hyo Jin Jo, Samuel Woo, Ji Young Chun, Jooyoung Park, and Dong Hoon Lee. Identifying ECUs Using Inimitable Characteristics of Signals in Controller Area Networks. *IEEE Trans. Vehicular Technology*, 67(6):4757–4770, 2018.

[40] A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[41] Lucian Cojocar, Jonas Zaddach, Roel Verdult, Herbert Bos, Aurélien Francillon, and Davide Balzarotti. PIE: Parser Identification in Embedded Systems. In *Proceedings of the 31st Annual Computer Security Applications Conference*, 2015.

[42] European Commission. Commission Directive 95/56/EC, Euratom of 8 November 1995 adapting to technical progress Council Directive 74/61/EEC relating to devices to prevent the unauthorized use of motor vehicles. *Official Journal L 286*, pages 1–44, 1995.

[43] Renesas Electronics Corporation. V850 JTAG OCD Checker. `https://www.renesas.com/us/en/doc/products/tool/doc/004/r20ut2462ej0200_v850_joc.pdf`. accessed: 2020/09/01.

[44] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. Inception: System-Wide Security Testing of Real-World Embedded Systems Software. In *Proceedings of the 27th USENIX Security Symposium*, 2018.

[45] Nassim Corteggiani and Aurélien Francillon. HardSnap: Leveraging hardware snapshotting for embedded systems security testing. In *Proceedings of the 50th IEEE/IFIP International Conference on Dependable Systems and Networks*, 2020.

[46] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A Large-Scale Analysis of the Security of Embedded Firmwares. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, 2014.

[47] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces. In *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security*, 2016.

[48] Nicolas Courtois, Gregory V. Bard, and David A. Wagner. Algebraic and Slide Attacks on KeeLoq. In Kaisa Nyberg, editor, *Fast Software Encryption, 15th International Workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, Revised Selected Papers*, volume 5086 of *Lecture Notes in Computer Science*, pages 97–115. Springer, 2008.

[49] Ang Cui and Rick Housley. BADFET: defeating modern secure boot using second-order pulsed electromagnetic fault injection. In William Enck and Collin Mulliner, editors, *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017*. USENIX Association, 2017.

[50] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[51] Franck de Goër, Sanjay Rawat, Dennis Andriesse, Herbert Bos, and Roland Groz. Now You See Me: Real-time Dynamic Function Call Detection. In *Proceedings of the 2018 Annual Computer Security Applications Conference*, 2018.

[52] Department for Transport. Transport Statistics Great Britain 2019. `https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/870647/tsgb-2019.pdf`.

[53] Digital Kaos. `https://www.digital-kaos.co.uk`.

[54] EU Directive. 98/69/EC of the European Parliament and of the Council of 13 October 1998 relating to measures to be taken against air pollution by emissions from motor vehicles and amending Council Directive 70/220/EEC. *Official Journal of the European Communities L*, 350(28):12, 1998.

[55] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable Reverse Engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, 2015.

[56] Domen Puncer Kugler. LPC13xx Bootloader Reverse Engineering. `https://github.com/domenpk/lpc13xx_boot_analysis`, Jan 2016.

[57] Thomas Eisenbarth, Timo Kasper, Amir Moradi, Christof Paar, Mahmoud Salmasizadeh, and Mohammad T. Manzuri Shalmani. On the Power of Power Analysis in the Real World: A Complete Break of the KeeLoqCode Hopping Scheme. In David A. Wagner, editor, *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, volume 5157 of *Lecture Notes in Computer Science*, pages 203–220. Springer, 2008.

[58] European Automobile Manufacturers Association. Average Vehicle Age. `https://www.acea.be/statistics/tag/category/average-vehicle-age`.

[59] European Automobile Manufacturers Association. World Motor Vehicle Production. `https://www.acea.be/statistics/article/world-production`.

[60] Bo Feng, Alejandro Mera, and Long Lu. P$^2$IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[61] Ford car market share in the United Kingdom (UK) from October 2016 to September 2020. `https://www.statista.com/statistics/300423/ford-car-market-share-in-the-united-kingdom/`, 2021.

[62] I. Foster, A. Prudhomme, K. Koscher, and S. Savage. Fast and Vulnerable: A Story of Telematic Failures. In *Proceedings of the 9th USENIX Conference on Offensive Technologies*. WOOT"15, 2015.

[63] Ian D. Foster and Karl Koscher. Exploring Controller Area Networks. *login Usenix Mag.*, 40(6), 2015.

[64] Aurélien Francillon and Claude Castelluccia. Code Injection Attacks on Harvard-Architecture Devices. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 15–26, 2008.

[65] Aurélien Francillon, Boris Danev, and Srdjan Capkun. Relay Attacks on Passive Keyless Entry and Start Systems in Modern Cars. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011.

[66] Jan Friebertshäuser. Polypyus – The Firmware Historian. `https://github.com/seemoo-lab/polypyus/`. accessed: 2020/07/25.

[67] Flavio D Garcia, Gerhard de Koning Gans, and Roel Verdult. Tutorial: Proxmark, the swiss army knife for rfid security research. Technical report, Radboud University Nijmegen, 2012.

[68] Flavio D Garcia, Gerhard de Koning Gans, and Roel Verdult. Wirelessly lockpicking a smart card reader. *International journal of information security*, 13(5):403–420, 2014.

[69] Flavio D. Garcia, Gerhard de Koning Gans, Roel Verdult, and Milosch Meriac. Dismantling iClass and iClass Elite. In *17th European Symposium on Research in Computer Security (ESORICS 2012)*, volume 7459 of *Lecture Notes in Computer Science*, pages 697–715. Springer-Verlag, 2012.

[70] Flavio D. Garcia, David Oswald, Timo Kasper, and Pierre Pavlidès. Lock It and Still Lose It - on the (In)Security of Automotive Remote Keyless Entry Systems. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, 2016.

[71] Chris Gerlinsky. Breaking Code Read Protection on the NXP LPC-family Microcontrollers. `https://recon.cx/2017/brussels/resources/slides/RECON-BRX-2017-Breaking_CRP_on_NXP_LPC_Microcontrollers_slides.pdf`.

[72] David Gessner, Manuel Barranco, Alberto Ballesteros, and Julian Proenza. sfiCAN: A Star-Based Physical Fault-Injection Infrastructure for CAN Networks. *IEEE Trans. Vehicular Technology*, 63(3):1335–1349, 2014.

[73] Brett Giller. Implementing Practical Electrical Glitching Attacks. `https://www.blackhat.com/docs/eu-15/materials/eu-15-Giller-Implementing-Electrical-Glitching-Attacks.pdf`.

[74] Christophe Giraud. DFA on AES. In Hans Dobbertin, Vincent Rijmen, and Aleksandra Sowa, editors, *Advanced Encryption Standard - AES, 4th International Conference, AES 2004, Bonn, Germany, May 10-12, 2004, Revised Selected and Invited*

*Papers*, volume 3373 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2004.

[75] Travis Goodspeed. Cracking the MSP430 BSL. `https://fahrplan.events.ccc.de/congress/2008/Fahrplan/attachments/1191_goodspeed_25c3_bslc.pdf`, 2008.

[76] Travis Goodspeed. A 16 Bit Rootkit, and Second Generation Zigbee Chips. `https://www.blackhat.com/presentations/bh-usa-09/GOODSPEED/BHUSA09-Goodspeed-ZigbeeChips-SLIDES.pdf`, 2009.

[77] Travis Goodspeed and Aurélien Francillon. Half-Blind Attacks: Mask ROM Bootloaders Are Dangerous. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, WOOT 2009, page 6, USA, 2009. USENIX Association.

[78] Sudhakar Govindavajhala and Andrew W. Appel. Using Memory Errors to Attack a Virtual Machine. In *2003 IEEE Symposium on Security and Privacy (S&P 2003), 11-14 May 2003, Berkeley, CA, USA*, pages 154–165. IEEE Computer Society, 2003.

[79] Bogdan Groza, Pal-Stefan Murvay, Anthony Van Herrewege, and Ingrid Verbauwhede. Libra-can: A lightweight broadcast authentication protocol for controller area networks. In *Cryptology and Network Security, 11th International Conference, CANS 2012, Darmstadt, Germany, December 12-14, 2012. Proceedings*, pages 185–200, 2012.

[80] Bogdan Groza, Lucian Popa, and Pal-Stefan Murvay. INCANTA - INtrusion Detection in Controller Area Networks with Time-Covert Authentication. In *Security and Safety Interplay of Intelligent Software Systems - ESORICS 2018 International Workshops, ISSA/CSITS@ESORICS 2018, Barcelona, Spain, September 6-7, 2018, Revised Selected Papers*, pages 94–110, 2018.

[81] Michael Gruhn and Tilo Müller. On the practicability of cold boot attacks. In *2013 International Conference on Availability, Reliability and Security, ARES 2013, Regensburg, Germany, September 2-6, 2013*, pages 390–397. IEEE Computer Society, 2013.

[82] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. In *International workshop on cryptographic hardware and embedded systems*, pages 119–132. Springer Berlin Heidelberg, 2004.

[83] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, and Giovanni Vigna. Toward the Analysis of Embedded Firmware through Automated Re-hosting. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses*, 2019.

[84] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In Paul C. van Oorschot, editor, *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, pages 45–60. USENIX Association, 2008.

[85] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[86] Oliver Hartkopp and R MaCAN SCHILLING. Message Authenticated CAN. In *Escar Conference, Berlin, Germany*, 2012.

[87] Ahmed Hazem and HA Fahmy. LCAP - A Lightweight CAN Authentication Protocol for Securing In-Vehicle Networks. In *10th escar Embedded Security in Cars Conference, Berlin, Germany*, volume 6, 2012.

[88] Hex-Rays. IDA Pro. accessed: 2020/07/27.

[89] Christopher Hicks, Flavio Garcia, and David Oswald. Dismantling the AUT64 Automotive Cipher. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2):46–69, May 2018.

[90] Michael Hutter and Jörn-Marc Schmidt. The Temperature Side Channel and Heating Fault Attacks. In Aurélien Francillon and Pankaj Rohatgi, editors, *Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*, volume 8419 of *Lecture Notes in Computer Science*, pages 219–235. Springer, 2013.

[91] Hyundai Motor Group's vehicle sales worldwide from 2000 to 2018, by brand. `https://www.statista.com/statistics/683556/hyundai-kia-vehicle-sales-worldwide/`, 2021.

[92] Sebastiaan Indesteege, Nathan Keller, Orr Dunkelman, Eli Biham, and Bart Preneel. A Practical Attack on KeeLoq. In *Annual International Conference on the*

*Theory and Applications of Cryptographic Techniques*, pages 1–18. Springer Berlin Heidelberg, 2008.

[93] ISO 14230-1:2012. Road vehicles – Diagnostic communication over K-Line (DoK-Line) – Part 1: Physical layer. Standard, International Organization for Standardization, Geneva, CH, 2012.

[94] Muhui Jiang, Yajin Zhou, Xiapu Luo, Ruoyu Wang, Yang Liu, and Kui Ren. An empirical study on ARM disassembly tools. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[95] Kyungho Joo, Wonsuk Choi, and Dong Hoon Lee. Hold the Door! Fingerprinting Your Car Key to Prevent Keyless Entry Car Theft. *CoRR*, abs/2003.13251, 2020.

[96] Marc Joye and Michael Tunstall, editors. *Fault Analysis in Cryptography.* Information Security and Cryptography. Springer, 2012.

[97] U Kaiser. A low-power digital signature transponder ic for high performance rfid authentication. In *Proceedings of European Conference on Circuit Theory and Design (ECCTD)*, volume 99, pages 45–48, 1999.

[98] Ulrich Kaiser. Digital Signature Transponder. In Paris Kitsos and Yan Zhang, editors, *RFID Security: Techniques, Protocols and System-on-Chip Design*, pages 177–189. Springer US, Boston, MA, 2008.

[99] Markus Kammerstetter, Markus Muellner, Daniel Burian, Christian Kudera, and Wolfgang Kastner. Computer Aided IC Reverse Engineering Methodology on the Basis of the TI DST80 Cryptographic Immobilizer Tag. *IEEE International Workshop on Physical Attacks and Inspection on Electronics (PAINE) Workshop 2018*, 2018.

[100] Markus Kasper, Timo Kasper, Amir Moradi, and Christof Paar. Breaking KeeLoq in a Flash: On Extracting Keys at Lightning Speed. In *Progress in Cryptology - AFRICACRYPT 2009, Second International Conference on Cryptology in Africa, Gammarth, Tunisia, June 21-25, 2009. Proceedings*, pages 403–420, 2009.

[101] Timo Kasper, David Oswald, and Christof Paar. A Versatile Framework for Implementation Attacks on Cryptographic RFIDs and Embedded Devices. *Trans. Comput. Sci.*, 10:100–130, 2010.

[102] Auguste Kerckhoffs. La cryptographie militaire. *Journal des sciences militaires 9*, pages 5–38, 1883.

[103] Daan Keuper and Thijs Alkemade. The Connected Car: Ways to get Unauthorized Access and Potential Implications. Technical report, Computest, 2018.

[104] J. Khan. ADvanced Encryption STAndard (ADESTA) for Diagnostics over CAN. *SAE International Journal of Passenger Cars - Electronic and Electrical Systems*, 8(2), 2015.

[105] Sultan Qasim Khan. Microcontroller readback protection: Bypasses and defenses. Technical report, nccgroup, 2020.

[106] James C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, July 1976.

[107] H Kleinknecht. Can Calibration Protocol Version 2.1. *Germany: ASAM eV*, pages 2–18, 1999.

[108] Marcel Kneib and Christopher Huth. On the Fingerprinting of Electronic Control Units Using Physical Characteristics in Controller Area Networks. In *47. Jahrestagung der Gesellschaft für Informatik, Informatik 2017, Chemnitz, Germany, September 25-29, 2017*, pages 875–882, 2017.

[109] Oliver Kömmerling and Markus G. Kuhn. Design Principles for Tamper-Resistant Smartcard Processors. In Scott B. Guthery and Peter Honeyman, editors, *Proceedings of the 1st Workshop on Smartcard Technology, Smartcard 1999, Chicago, Illinois, USA, May 10-11, 1999.* USENIX Association, 1999.

[110] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, et al. Experimental Security Analysis of a Modern Automobile. In *2010 IEEE Symposium on Security and Privacy (SP)*, pages 447–462. Institute of Electrical and Electronics Engineers, 2010.

[111] Karl Koscher, Tadayoshi Kohno, and David Molnar. SURROGATES: Enabling near-real-time dynamic analyses of embedded systems. In *Proceedings of the 9th USENIX Workshop on Offensive Technologies*, 2015.

[112] Jonas Krautter, Dennis R. E. Gnad, and Mehdi Baradaran Tahoori. FPGAhammer: Remote Voltage Fault Attacks on Shared FPGAs, suitable for DFA on AES. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):44–68, 2018.

[113] Ryo Kurachi, Yutaka Matsubara, Hiroaki Takada, Naoki Adachi, Yukihiro Miyashita, and Satoshi Horihata. CaCAN - Centralized Authentication System in CAN (Controller Area Network). In *14th Int. Conf. on Embedded Security in Cars (ESCAR 2014)*, 2014.

[114] Tencent Keen Security Lab. Experimental Security Research of Tesla Autopilot. Technical report, Tencent Keen Security, 2019.

[115] Adam Laurie. Atmel SAM7XC Crypto Co-Processor key recovery. `https://adamsblog.rfidiot.org/2013/02/atmel-sam7xc-crypto-co-processor-key.html`.

[116] LimitedResults. Pwn the ESP32 Forever: Flash Encryption and Sec. Boot Keys Extraction. `https://limitedresults.com/2019/11/pwn-the-esp32-forever-flash-encryption-and-sec-boot-keys-extraction/`, November 2019.

[117] Paul Lipschutz. Control Device for Vehicle Locks, 1978. `https://patents.google.com/patent/US4258352A/`.

[118] Yifan Lu. Injecting Software Vulnerabilities with Voltage Glitching. *CoRR*, abs/1903.08102, 2019.

[119] Dominik Maier, Lukas Seidel, and Shinjo Park. BaseSAFE: Baseband Sanitized Fuzzing through Emulation. In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2020.

[120] Patricia Mayhew and Great Britain. *Crime as opportunity*, volume 34. HM Stationery Office London, 1976.

[121] Alejandro Mera, Bo Feng, Long Lu, Engin Kirda, and William Robertson. Dice: Automatic emulation of dma input channels for dynamic firmware analysis. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, 2021.

[122] Milosch Meriac. Heart of Darkness - exploring the uncharted backwaters of HID iCLASS security. Technical report, Bitmanufaktur GmbH, 2010.

[123] MHH Auto. `https://mhhauto.com/`.

[124] Alyssa Milburn, Niek Timmers, Nils Wiersma, Ramiro Pareja, and Santiago Cordoba. There Will Be Glitches: Extracting and Analyzing Automotive Firmware Efficiently. *Black Hat USA*, 2018.

[125] Charlie Miller and Chris Valasek. Adventures in Automotive Networks and Control Units. *Def Con*, 21:260–264, 2013.

[126] Charlie Miller and Chris Valasek. Remote Exploitation of an Unaltered Passenger Vehicle. *Black Hat USA*, 2015:91, 2015.

[127] Chris Miller and Chris Valasek. Car Hacking: For Poories. Technical report, IOActive Report, 2015.

[128] Nick Morgan, Oliver Shaw, Andy Feist, and Christos Byron. *Reducing criminal opportunity: vehicle security and vehicle crime.* Home Office, 2016.

[129] MSP430 Programming With the JTAG Interface. `https://www.ti.com/lit/ug/slau320ai/slau320ai.pdf?ts=1588202726767`.

[130] Marius Muench. *Dynamic Binary Firmware Analysis: Challenges and Solutions.* PhD thesis, Thesis, 09 2019.

[131] Marius Muench, Aurélien Francillon, and Davide Balzarotti. Avatar$^2$: A multi-target orchestration platform. In *Workshop on Binary Analysis Research*, 2018.

[132] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *Proceedings of the 2018 Network and Distributed System Security Symposium*, 2018.

[133] Tilo Müller and Michael Spreitzenbarth. FROST - Forensic Recovery of Scrambled Telephones. In *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings*, volume 7954 of *Lecture Notes in Computer Science*, pages 373–388. Springer, 2013.

[134] PEMicro Multilink. available online at `http://www.pemicro.com/products/product_viewDetails.cfm?product_id=15320168&productTab=1`, 2021.

[135] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[136] Pal-Stefan Murvay and Bogdan Groza. Source Identification Using Signal Characteristics in Controller Area Networks. *IEEE Signal Process. Lett.*, 21(4):395–399, 2014.

[137] Pal-Stefan Murvay and Bogdan Groza. DoS Attacks on Controller Area Networks by Fault Injections from the Software Layer. In *Proceedings of the 12th International Conference on Availability, Reliability and Security, Reggio Calabria, Italy, August 29 - September 01, 2017*, pages 71:1–71:10, 2017.

[138] Pal-Stefan Murvay and Bogdan Groza. Practical Security Exploits of the FlexRay In-Vehicle Communication Protocol. In *Risks and Security of Internet and Systems - 13th International Conference, CRiSIS 2018, Arcachon, France, October 16-18, 2018, Revised Selected Papers*, pages 172–187, 2018.

[139] Pal-Stefan Murvay and Bogdan Groza. TIDAL-CAN: Differential Timing Based Intrusion Detection and Localization for Controller Area Network. *IEEE Access*, 8:68895–68912, 2020.

[140] Roger M Needham and David J Wheeler. Tea extensions. Technical report, Cambride University, 1997.

[141] Sen Nie, Ling Liu, and Yuefeng Du. Free-Fall: Hacking Tesla from Wireless to CAN Bus. *Black Hat USA*, 2017.

[142] Thomas Nolte, Hans Hansson, Christer Norström, and Sasikumar Punnekkat. Using bit-stuffing distributions in can analysis. In *IEEE Real-Time Embedded Systems Workshop at the Real-Time Systems Symposium*, 2001.

[143] Stefan Nürnberger and Christian Rossow. - vatican - vetted, authenticated CAN bus. In *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, pages 106–124, 2016.

[144] NXP. AN10968: Using Code Read Protection in LPC1100 and LPC1300. `https://www.nxp.com/docs/en/application-note/AN10968.pdf`.

[145] NXP. UM10375: LPC1311/13/42/43 User manual. `https://www.nxp.com/docs/en/user-guide/UM10375.pdf`.

[146] Introduction to HCS08 Background Debug Mode. `https://www.nxp.com/docs/en/application-note/AN3335.pdf`.

[147] MPC5500 Flash Programming Through Nexus/JTAG. `https://www.nxp.com/docs/en/application-note/AN3283.pdf`.

[148] Johannes Obermaier, Marc Schink, and Kosma Moczek. One Exploit to Rule them All? On the Security of Drop-in Replacement and Counterfeit Microcontrollers. In Yuval Yarom and Sarah Zennou, editors, *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*. USENIX Association, 2020.

[149] Johannes Obermaier and Stefan Tatschner. Shedding too much Light on a Microcontroller's Firmware Protection. In *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017*, 2017.

[150] Colin O'Flynn. Fault injection using crowbars on embedded systems. *IACR Cryptol. ePrint Arch.*, 2016:810, 2016.

[151] Colin O'Flynn. BAM BAM!! On Reliability of EMFI for in-situ Automotive ECU Attacks. *IACR Cryptol. ePrint Arch.*, 2020:937, 2020.

[152] Colin O'Flynn and Zhizhang (David) Chen. ChipWhisperer: An Open-Source Platform for Hardware Embedded Security Research. In *Constructive Side-Channel Analysis and Secure Design - 5th International Workshop, COSADE 2014, Paris, France, April 13-15, 2014. Revised Selected Papers*, pages 243–260, 2014.

[153] OpenOCD. available online at `https://sourceforge.net/projects/openocd/`, 2020.

[154] David Oswald. Generic Implementation ANalysis Toolkit. available online at `https://sourceforge.net/projects/giant`, 2016.

[155] Andrea Palanca, Eric Evenchick, Federico Maggi, and Stefano Zanero. A Stealth, Selective, Link-Layer Denial-of-Service Attack Against Automotive Networks. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings*, pages 185–206, 2017.

[156] Percepio. Tracealyzer. `https://percepio.com/tracealyzer/`. accessed: 2020/07/25.

[157] Mert D. Pesé, Troy Stacer, C. Andrés Campos, Eric Newberry, Dongyao Chen, and Kang G. Shin. LibreCAN: Automated CAN Message Translator. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 2283–2300, 2019.

[158] Stjepan Picek, Lejla Batina, Domagoj Jakobovic, and Rafael Boix Carpi. Evolving genetic algorithms for fault injection attacks. In *37th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2014, Opatija, Croatia, May 26-30, 2014*, pages 1106–1111, 2014.

[159] Thomas Pornin. Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). RFC 6979, RFC Editor, 2013.

[160] q3k. Renesas M16C programmer. `https://github.com/q3k/m16c-interface`, July 2017.

[161] J-J Quisquater. Eddy current for magnetic analysis with active sensor. *Proceedings of Esmart, 2002*, pages 185–194, 2002.

[162] Jean-Jacques Quisquater and David Samyde. ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards. In Isabelle Attali and Thomas P. Jensen, editors, *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001, Proceedings*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2001.

[163] Andreea-Ina Radu and Flavio D Garcia. LeiA: A Lightweight Authentication Protocol for CAN. In *21st European Symposium on Research in Computer Security (ESORICS 2016)*, pages 283–300. Springer-Verlag, 2016.

[164] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. KARONTE: Detecting Insecure Multi-binary Interactions in Embedded Firmware. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, 2020.

[165] EUROPA Press releases. Improving the safety and environmental performance of vehicles, 2008.

[166] 78K0/Kx2 Flash Memory Programming. https://www.renesas.com/eu/en/doc/DocumentServer/024/U17739EJ3V0AN00.pdf.

[167] 78K0/Kx2 User's Manual: Hardware. https://www.renesas.com/in/en/doc/DocumentServer/011/r01uh0008ej0401_78k0kx2.pdf.

[168] Code Flash Libraries (Flash Self Programming Libraries). https://www.renesas.com/eu/en/products/software-tools/tools/self-programming-library/code-flash-libraries.html.

[169] Thomas Roth, Dmitry Nedospasov, and Josh Datko. wallet.fail. https://fahrplan.events.ccc.de/congress/2018/Fahrplan/events/9563.html.

[170] Ishtiaq Rouf, Robert D. Miller, Hossen A. Mustafa, Travis Taylor, Sangho Oh, Wenyuan Xu, Marco Gruteser, Wade Trappe, and Ivan Seskar. Security and Privacy Vulnerabilities of In-Car Wireless Networks: A Tire Pressure Monitoring System Case Study. In *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*, pages 323–338. USENIX Association, 2010.

[171] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[172] D. Samyde, S. Skorobogatov, R. Anderson, and J. . Quisquater. On a new way to read data from memory. In *First International IEEE Security in Storage Workshop, 2002. Proceedings.*, pages 65–69, 2002.

[173] Marc Schink and Johannes Obermaier. Exception(al) Failure - Breaking the STM32F1 Read-Out Protection. https://blog.zapb.de/stm32f1-exceptional-failure/.

[174] Marc Schink and Johannes Obermaier. Taking a Look into Execute-Only Memory. In Alex Gantman and Clémentine Maurice, editors, *13th USENIX Workshop on Offensive Technologies, WOOT 2019, Santa Clara, CA, USA, August 12-13, 2019.* USENIX Association, 2019.

[175] SEGGER. SystemView. https://www.segger.com/products/development-tools/systemview/. accessed: 2020/07/25.

[176] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[177] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings of the 2015 Network and Distributed System Security Symposium*, 2015.

[178] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 138–157, 2016.

[179] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, 2016.

[180] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. Rise of the HaCRS: Augmenting Autonomous Cyber Reasoning Systems with Human Assistance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[181] Sergei Skorobogatov. Low temperature data remanence in static RAM. Technical report, University of Cambridge, Computer Laboratory, 2002.

[182] Sergei Skorobogatov. Flash Memory 'Bumping' Attacks. In *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, pages 158–172, 2010.

[183] Sergei Skorobogatov. Optical Fault Masking Attacks. In Luca Breveglieri, Marc Joye, Israel Koren, David Naccache, and Ingrid Verbauwhede, editors, *2010 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2010, Santa Barbara, California, USA, 21 August 2010*, pages 23–29. IEEE Computer Society, 2010.

[184] Sergei P. Skorobogatov. Copy Protection in Modern Microcontrollers. `https://www.cl.cam.ac.uk/~sps32/mcu_lock.html`.

[185] Sergei P. Skorobogatov. Data Remanence in Flash Memory Devices. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 339–353. Springer, 2005.

[186] Sergei P. Skorobogatov and Ross J. Anderson. Optical Fault Induction Attacks. In *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, pages 2–12, 2002.

[187] Sergei Petrovich Skorobogatov. *Semi-invasive attacks: A new approach to hardware security analysis*. PhD thesis, University of Cambridge, 2005.

[188] Dieter Spaar. Beemer, Open Thyself! – Security vulnerabilities in BMW's ConnectedDrive . `https://www.heise.de/ct/artikel/Beemer-Open-Thyself-Security-vulnerabilities-in-BMW-s-ConnectedDrive-2540957.html`.

[189] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard Shrobe, and Mathias Payer. FirmFuzz: Automated IoT Firmware Introspection and Analysis. In *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*, 2019.

[190] STM32 Microcontroller Debug Toolbox. `http://www.st.com/resource/en/application_note/dm00354244-stm32-microcontroller-debug-toolbox-stmicroelectro pdf`. accessed: 2020/09/01.

[191] ST Microelectronics. STM8 bootloader. `https://www.st.com/content/ccc/resource/technical/document/user_manual/e4/83/c1/d6/ee/d8/49/b8/CD00201192.pdf/files/CD00201192.pdf/jcr:content/translations/en.CD00201192.pdf`.

[192] STM8 SWIM communication protocol and debug module. `https://www.st.com/content/ccc/resource/technical/document/user_manual/ca/89/41/4e/72/31/49/f4/CD00173911.pdf/files/CD00173911.pdf/jcr:content/translations/en.CD00173911.pdf`.

[193] STM8AF Series. `https://www.st.com/content/ccc/resource/technical/document/reference_manual/9a/1b/85/07/ca/eb/4f/dd/CD00190271.pdf/files/CD00190271.pdf/jcr:content/translations/en.CD00190271.pdf`.

[194] ST Microelectronics. *STM8L151x4, STM8L151x6, STM8L152x4, STM8L152x6 datasheet.*

[195] Daehyun Strobel, David Oswald, Bastian Richter, Falk Schellenberg, and Christof Paar. Microcontrollers as (In)Security Devices for Pervasive Computing Applications. *Proceedings of the IEEE*, 102(8):1157–1173, 2014.

[196] Samuel Junjie Tan, Sergey Bratus, and Travis Goodspeed. Interrupt-oriented Bugdoor Programming: A Minimalist Approach to Bugdooring Embedded Systems Firmware. In *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014.

[197] Katherine Temkin. Vulnerability Disclosure: Fusée Gelée. `https://github.com/Qyriad/fusee-launcher/blob/master/report/fusee_gelee.md`.

[198] Exploiting Wi-Fi Stack on Tesla Model S. `https://keenlab.tencent.com/en/2020/01/02/exploiting-wifi-stack-on-tesla-model-s/`, 2020.

[199] Tencent Keen Security Lab: Experimental Security Assessment on Lexus Cars . `https://keenlab.tencent.com/en/2020/03/30/Tencent-Keen-Security-Lab-Experimental-Security-Assessment-on-Lexus-Cars/`, 2020.

[200] Texas Instruments. *Digital Signal Transponder With DST80 Authentication, EEPROM, and LF Immobilizer*, 2012.

[201] Texas Instruments. *TMS3705 Transponder Base Station IC*, 2012.

[202] Texas Instruments. *TMS37126 - Remote access identification device with integrated 3D wakeup receiver and immobilizer interface*, 2012.

[203] Texas Instruments. *TMS37F128 - Controller Remote Access Identification (CRAID) with integrated microcontroller, 3D wakeup receiver, and immobilizer interface*, 2012.

[204] Sam L. Thomas, Tom Chothia, and Flavio D. Garcia. Stringer: Measuring the Importance of Static Data Comparisons to Detect Backdoors and Undocumented Functionality. In *Proceedings of the 22nd European Symposium on Research in Computer Security*, 2017.

[205] Sam L. Thomas, Flavio D. Garcia, and Tom Chothia. HumIDIFy: A Tool for Hidden Functionality Detection in Firmware. In *Proceedings of the 14th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2017.

[206] Toyota Transponder Chip Key & Remote Key Fob Programming Instructions. http://www.diy-time.com/automotive/toyota/program-chip-toyota-key-remote-fob/.

[207] Transpondery. http://www.transpondery.com/transponder_catalog/toyota_transponder_catalog.html.

[208] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault. In Claudio A. Ardagna and Jianying Zhou, editors, *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication*, pages 224–233. Springer, 2011.

[209] (US) National Security Agency (NSA). Ghidra. accessed: 2020/07/27.

[210] C. Valasek and C. Miller. Remote Exploitation of an Unaltered Passenger Vehicle. Technical report, Illmatics, 2015.

[211] Anthony Van Herrewege, Dave Singelee, and Ingrid Verbauwhede. CANAuth - A Simple, Backward Compatible Broadcast Authentication Protocol for CAN bus. In *ECRYPT Workshop on Lightweight Cryptography*, volume 2011, 2011.

[212] Jasper G. J. van Woudenberg, Marc F. Witteman, and Federico Menarini. Practical Optical Fault Injection on Secure Microcontrollers. In Luca Breveglieri, Sylvain Guilley, Israel Koren, David Naccache, and Junko Takahashi, editors, *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2011, Tokyo, Japan, September 29, 2011*, pages 91–99. IEEE Computer Society, 2011.

[213] Sebastian Vasile, David Oswald, and Tom Chothia. Breaking All the Things - A Systematic Survey of Firmware Extraction Techniques for IoT Devices. In *Smart Card Research and Advanced Applications, 17th International Conference, CARDIS 2018, Montpellier, France, November 12-14, 2018, Revised Selected Papers*, pages 171–185, 2018.

[214] Vector Informatik. *Product Catalog*, 5 2010.

[215] Roel Verdult, Flavio D Garcia, and Josep Balasch. Gone in 360 seconds: Hijacking with Hitag2. In *21st USENIX Security Symposium (USENIX Security 2012)*, pages 237–252. USENIX Association, 2012.

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

[217] Aram Verstegen, Roel Verdult, and Wouter Bokslag. Hitag 2 Hell - Brutally Optimizing Guess-and-Determine Attacks. In *12th USENIX Workshop on Offensive Technologies, WOOT 2018, Baltimore, MD, USA, August 13-14, 2018*, 2018.

[218] Volvo Trucks North America Inc . Three-point Seat Belt , 1986. `https://patents.google.com/patent/US5020856A/en`.

[219] Qiyan Wang and Sanjay Sawhney. VeCure: A Practical Security Framework to Protect the CAN Bus of Vehicles. In *4th International Conference on the Internet of Things, IOT 2014, Cambridge, MA, USA, October 6-8, 2014*, pages 13–18, 2014.

[220] Haohuang Wen, Qingchuan Zhao, Qi Alfred Chen, and Zhiqiang Lin. Automated Cross-Platform Reverse Engineering of CAN Bus Commands From Mobile Apps. In *Proceedings 2020 Network and Distributed System Security Symposium (NDSS'20)*, 2020.

[221] Jared Wiltshire. VW Transport Protocol 2.0 (TP 2.0) for CAN bus. `https://jazdw.net/tp20`.

[222] Lennert Wouters, Eduard Marin, Tomer Ashur, Benedikt Gierlichs, and Bart Preneel. Fast, Furious and Insecure: Passive Keyless Entry and Start Systems in Modern Supercars. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):66–85, 2019.

[223] ASAM MCD-1 XCP. The Universal Measurement and Calibration Protocol Family. Standard, Association of Standardisation and Automation and Measuring Systems, 2016.

[224] Qing Yang and Lin Huang. *Inside Radio: An Attack and Defense Guide.* Springer, 2018.

[225] Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation. *J. Hardware and Systems Security*, 2(2):111–130, 2018.

[226] Jonas Zaddach, Luca Bruno, Aurélien Francillon, and Davide Balzarotti. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Proceedings of the 2014 Network and Distributed System Security Symposium*, 2014.

[227] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *Proceedings of the 28th USENIX Security Symposium*, 2019.

[228] Tobias Ziermann, Stefan Wildermann, and Jürgen Teich. CAN+: A new Backward-compatible Controller Area Network (CAN) protocol with up to 16× higher data rates. In *Design, Automation and Test in Europe, DATE 2009, Nice, France, April 20-24, 2009*, pages 1088–1093, 2009.