

## MASTER

### An assessment of ECM authentication in modern vehicles

Bokslag, W.

*Award date:*  
2017

[Link to publication](#)

#### **Disclaimer**

This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

#### **General rights**

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

# An assessment of ECM authentication in modern vehicles

*Master Thesis*

Wouter Bokslag

Supervisor:  
prof. dr. Sandro Etalle

Tutor:  
dr. ing. Roel Verdult

Assessor:  
dr. Benne de Weger

CONFIDENTIAL

Eindhoven, July 2017



# Abstract

All modern road vehicles are fitted with an electronic immobilization system, intended to prevent the vehicle from starting unless an authorized transponder is presented. Many immobilization solutions are on the market, and extensive research into the quality of these solutions has been done. However, internally, vehicles generally use a different protocol for authentication between the car immobilizer component and the engine control module. Insecurities in these authentication protocols would potentially allow attackers to bypass the immobilization system. This thesis presents an analysis of three such authentication protocols, extracted from vehicles from three different manufacturers. Using a standardized connector, CAN-bus traffic was analysed, and by means of reverse engineering, the underlying cryptographic primitives were identified and assessed. Two solutions were cryptographically and practically broken, while the third solution does not contain obvious cryptographic weaknesses.



# Contents

<b>Contents</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	1
1.2 Research questions . . . . .	2
1.3 Structure . . . . .	2
<b>2 Notation and terminology</b>	<b>3</b>
<b>3 Standardized interfaces and protocols</b>	<b>5</b>
3.1 The SAE J1962 connector . . . . .	5
3.2 The ISO 9141 K-line protocol . . . . .	6
3.3 The ISO 15765 CAN protocol . . . . .	6
3.4 The ISO 14229 UDS protocol . . . . .	7
<b>4 Literature review</b>	<b>9</b>
<b>5 Methodology</b>	<b>11</b>
<b>6 Assessed vehicles</b>	<b>15</b>
6.1 Model A . . . . .	15
6.2 Model B . . . . .	15
6.3 Model C . . . . .	15
<b>7 Case study: Model A</b>	<b>16</b>
7.1 Identifying the protocol messages . . . . .	16
7.2 Obtaining the algorithm . . . . .	18
7.3 Algorithm details . . . . .	19
7.4 Properties of the cipher . . . . .	21
7.4.1 Insufficiently large keyspace . . . . .	21
7.4.2 Lack of diffusion . . . . .	21
7.4.3 Inverse of the transformation function . . . . .	21
7.4.4 Biased responses . . . . .	22
7.4.5 Leakage of key information . . . . .	22
7.5 Attacks . . . . .	22
7.5.1 Attack with valid key . . . . .	22
7.5.2 Car-only attack . . . . .	23
<b>8 Case study: Model B</b>	<b>24</b>
8.1 Identifying the protocol messages . . . . .	24
8.2 Obtaining the algorithm . . . . .	25
8.3 Algorithm details . . . . .	26
8.3.1 Phase 1: Initialization . . . . .	26

8.3.2	Phase 2: Proof generation . . . . .	27
8.3.3	Phase 3: Second secret absorption . . . . .	28
8.3.4	Phase 4: Response generation . . . . .	28
8.4	Properties of the cipher . . . . .	29
8.4.1	The feedback function . . . . .	29
8.4.2	Inverse round function . . . . .	29
8.4.3	Knowledge of state bits based on output bit . . . . .	30
8.5	Attacks . . . . .	30
8.5.1	Naive exhaustive-search . . . . .	30
8.5.2	Pruning exhaustive-search . . . . .	30
8.5.3	State reconstruction . . . . .	31
8.5.4	Deriving the second secret $l$ . . . . .	33
<b>9</b>	<b>Case study: Model C</b>	<b>34</b>
9.1	Identifying the protocol messages . . . . .	34
9.2	Obtaining the algorithm . . . . .	35
9.3	Algorithm origin . . . . .	36
9.4	Algorithm details . . . . .	39
9.4.1	Phase 1: Initialization . . . . .	39
9.4.2	Phase 2: Partial challenge absorption . . . . .	39
9.4.3	Phase 3: Partial response generation . . . . .	41
9.4.4	Phase 4: Remaining challenge absorption . . . . .	42
9.4.5	Phase 5: Remainder of response generation . . . . .	43
9.5	Properties of the cipher . . . . .	43
9.5.1	Table lookups . . . . .	43
9.5.2	Dependency between table lookups . . . . .	44
9.5.3	Inverse round function . . . . .	45
9.5.4	Rotate, round counts and array sizes . . . . .	46
9.5.5	Two-phase challenge absorption / response generation . . . . .	47
9.6	Attacks . . . . .	47
9.6.1	Divide-and-conquer attack . . . . .	47
9.6.2	Correlation attack . . . . .	47
9.6.3	Guess-and-determine attack . . . . .	48
9.6.4	Differential cryptanalysis . . . . .	48
9.6.5	Algebraic attacks . . . . .	48
9.6.6	Meet-in-the-middle attacks . . . . .	49
9.7	Remarks . . . . .	49
9.7.1	Usage in 32-bit mode . . . . .	49
9.7.2	Random number generation . . . . .	49
<b>10</b>	<b>Suggestions for improvement</b>	<b>51</b>
<b>11</b>	<b>Discussion</b>	<b>53</b>
11.1	How do manufacturers implement BCM-ECM authentication . . . . .	54
11.2	What is the strength of the cryptographic components used in the BCM-ECM authentication . . . . .	54
11.3	How can manufacturers improve upon the current strength of BCM-ECM authentication . . . . .	54
<b>12</b>	<b>Conclusions</b>	<b>56</b>
	<b>Bibliography</b>	<b>57</b>
	<b>Appendix</b>	<b>60</b>

<b>A</b>	<b>Model B lookup tables</b>	<b>60</b>
A.1	FeedbackTable . . . . .	60
A.2	OutputTable . . . . .	60
<b>B</b>	<b>Model C PCF7935 lookup table</b>	<b>61</b>
B.1	LookupTable . . . . .	61

# Chapter 1

## Introduction

### 1.1 Background

Nowadays, immobilizers play an essential role in preventing large-scale theft of vehicles. Intended to raise the complexity involved with stealing a vehicle by introducing non-mechanical safety measures, immobilizers have always worked by the same basic principle: disallowing the ignition<sup>1</sup> to activate until some secret is presented to the vehicle. Electronic car immobilizers have existed for as long as since 1919<sup>2</sup>, however, until 1990, only very few cars were equipped with electronic immobilization solutions<sup>3</sup>. Widespread adoption was a consequence of legislation: the European Union adopted regulation in 1995[7], mandating the use of electronic immobilization solutions in all cars sold within the EU as of 1998.

Immobilizers prove to be highly effective in the effort to reduce theft rates. According to a report in The Economic Journal, titled "The Engine Immobiliser: A Non-starter for Car Thieves" and written by Van Ours and Vollaard[40], the broad deployment of immobilization devices has led to a reduction in car theft of an estimated 40% on average during 1995-2008<sup>4</sup>. However, car thieves have proven to be able to bypass electronic security mechanisms[1]. The German motoring association ADAC maintains a list[2] of Keyless Go-equipped<sup>5</sup> vehicles they have tested for relay attack vulnerability. Nearly all vehicles on the list are affected. Also, multiple sources report cars being stolen by exploiting vulnerabilities in electronic security systems[38], sometimes to extents where insurance companies refuse to insure models unless additional security measures are taken[22]. The strength of these systems is thus crucial in protecting vehicles against unauthorized access and theft.

Most immobilizer systems are designed as follows. The mechanical car key is fitted with a small passive RF tag, referred to as the *transponder*. When the ignition is switched on, an antenna near the ignition lock emits a low frequency 125KHz radio field that powers the tag, and two-way communication between transponder and steering column is initiated. The car's immobilizer component (nowadays integrated in the BCM, or Body Control Module) queries the key and ascertains the transponder is authorized. Keyless Go systems operate in a similar manner, except that the key fob only houses a transponder, and the mechanical key is no longer required. Instead, the presence of the transponder is detected whenever the key is in proximity of the vehicle. As soon as the ignition is switched on, the ECM (or Engine Control Module) sends a challenge to the BCM. If the BCM is able to successfully authenticate with the transponder in the car key, the response to the ECM challenge will be computed and transmitted. If the ECM indeed receives a valid response, it will unlock, allowing the car to start.

---

<sup>1</sup>And often, also the fuel pump and starter circuit.

<sup>2</sup>In 1919, Evans et al. patented the first electric car immobilization solution[9], however primitive it was.

<sup>3</sup>According to [40], < 1 %.

<sup>4</sup>In the 2011 report, Van Ours and Vollaard state that during the ten years after the regulation went into effect, the overall rate of car theft dropped by 70% and 80% in the Netherlands and England/Wales respectively.

<sup>5</sup>Keyless Go is a technology where the key does not need to be inserted into the ignition lock.

## 1.2 Research questions

This thesis focuses on the protocols and underlying cryptographic primitives manufacturers employ for the ECM authentication step, in which the BCM authenticates towards the ECM. To the best of our knowledge, no academic research has explicitly targeted this part of the immobilizer system, while its security is as important as the strength of the transponder authentication. In order to properly assess the strength of inter-component authentication, we have restricted the scope to answer the following research questions.

- Which protocols do manufacturers use for BCM-ECM authentication?
- What is the strength of the cryptographic components used in BCM-ECM authentication?
- How can manufacturers improve upon the current strength of BCM-ECM authentication?

The first question aims to clarify the high-level approach to ECM authentication: which data is transmitted, whether one-way or mutual authentication protocols are used, which party initiates communication, and whether rate limiting or other mechanisms are implemented. The second question targets the cryptographic strength of the primitives used in the protocols, while the third question seeks to provide suggestions for improvements that will enhance the security of the ECM authentication step.

## 1.3 Structure

In order to answer the research questions, we first define some terminology and notations in Section 2. This is followed by a general introduction to the CAN bus, the standardized interface used for inter-component communication in nearly all recent<sup>6</sup> vehicles in Section 3. Related work will be discussed in Section 4, followed by a section outlining the general methodology adopted when assessing a vehicle. A motivation for the choice of models to investigate, as well as some details about the models is found in Section 6. In Sections 7, 8 and 9, the three case studies will be presented. This is followed by a section suggesting how manufacturers could adopt safe ECM authentication protocols. Section 11 will discuss the implications of the findings, followed by the conclusions, presented in Section 12.

---

<sup>6</sup>There *is* another standardized interface. K-line is the predecessor of CAN and may still be found in some, often older, vehicles. However, its popularity is declining and the interface is not relevant to the contents of this thesis.

## Chapter 2

# Notation and terminology

This section will briefly introduce the terminology and notations that will be used throughout the thesis.

ECU	Electronic Control Unit. This is the generic term for any embedded computer system in a vehicle. The term ECU is also commonly used for Engine Control Unit, which leads to a confusing ambiguity. In this thesis, ECU will always refer to the generic term, while ECM is used as an abbreviation for Engine Control Module.
ECM	Engine Control Module. This component is responsible for controlling the engine. It is also part of the immobilizer system, refusing to start the engine if the authenticity of the key has not been established.
BCM	Body Control Module, also called central electronics. This component controls many sensors and components not directly related to the engine. It authenticates towards the ECM in modern immobilizer systems.
CAN	Controller Area Network, the bus standard that is used in nearly all modern vehicles.
K-line	Predecessor of CAN, which resembles the UART (serial port) protocol.
UDS	Unified Diagnostic Services, as defined in ISO 14229. Defines a set of functions a vehicle should support, as well as their respective parameters and response packets.
Seedkey	Authentication method for diagnostic services over CAN. Before certain UDS functions may be used, a successful seedkey authentication must take place. The vehicle will present a challenge, the so called <i>seed</i> , to which a correct response ( <i>key</i> ) must be provided.
OBD-II	On Board Diagnostics II, an extension of the OBD standard. OBD-II has defined the capabilities of modern vehicles to self-diagnose and report error codes (in a partially standardized way) to compatible diagnostic tools. These tools can connect to the vehicle using the OBD-II connector, present in any modern vehicle.
ISO-TP	ISO standard that defines the structure of both single- and multi-frame CAN packets.
PIN	The car PIN, or vehicle security code, is a code that is required in order to perform key programming and other actions for which authorization is required. The terms PIN and vehicle security code are used interchangeably throughout the thesis.

The algorithms discussed perform both bit-wise and byte-wise operations. In order to clearly distinguish references to a single bit from references to a byte, the following notation is adopted.

- $r[n]$  refers to byte  $n$  in byte array  $r$ .
- $r_n$  refers to bit  $n$  in variable  $r$ . We follow the lsb 0 convention, thus,  $r_0$  is the least significant bit of the register.
- Concatenation of bits is denoted by comma-separating the bits, surrounded by square brackets:  $[b_k, b_l, b_m]$  is a 3-bit string with  $b_m$  as least significant bit.
- Concatenation of bytes is denoted as follows:  $r[n : m] = r[n], r[n + 1], \dots, r[m]$

- Any multi-byte variable  $r$  is in big-endian notation. That is,  $r[0]$  is the most significant byte.

We also define the behaviour of several bitwise operators. We define these operators over bit strings in  $\mathbb{F}_2^n$  for  $n > 0$ .

**Definition 2.0.1.** The bitwise **or** operator over bit strings  $\mathbb{F}_2^n \vee \mathbb{F}_2^n \rightarrow \mathbb{F}_2^n$  is defined as follows.

$$x \vee y = [(x_{n-1} \vee y_{n-1}), \dots, (x_0 \vee y_0)]$$

The bitwise **and** operator over bit strings  $\mathbb{F}_2^n \wedge \mathbb{F}_2^n \rightarrow \mathbb{F}_2^n$  is defined as follows.

$$x \wedge y = [(x_{n-1} \wedge y_{n-1}), \dots, (x_0 \wedge y_0)]$$

The bitwise **xor** operator over bit strings  $\mathbb{F}_2^n \oplus \mathbb{F}_2^n \rightarrow \mathbb{F}_2^n$  is defined as follows.

$$x \oplus y = ((x_{n-1} \oplus y_{n-1}), \dots, (x_0 \oplus y_0))$$

The bitwise **negate** operator over bit strings  $\overline{\mathbb{F}_2^n} \rightarrow \mathbb{F}_2^n$  is defined as follows.

$$\overline{x} = [\neg x_{n-1}, \dots, \neg x_0]$$

The bitwise **right shift** operator  $\mathbb{F}_2^n \gg \mathbb{N} \rightarrow \mathbb{F}_2^n$  is defined as

$$x \gg c = \begin{cases} x & \text{if } c = 0 \\ [0, x_{n-1}, \dots, x_1] & \text{if } c = 1 \\ (x \gg (n-1)) \gg 1 & \text{if } c > 1 \end{cases}$$

The bitwise **left shift** operator  $\mathbb{F}_2^n \ll \mathbb{N} \rightarrow \mathbb{F}_2^n$  is defined as

$$x \ll c = \begin{cases} x & \text{if } c = 0 \\ [x_{n-2}, \dots, x_0, 0] & \text{if } c = 1 \\ (x \ll (n-1)) \ll 1 & \text{if } c > 1 \end{cases}$$

The bitwise **right rotate** function  $\text{ror} : \mathbb{F}_2^n, \mathbb{N} \rightarrow \mathbb{F}_2^n$  is defined as

$$\text{ror}(x, c) = \begin{cases} x & \text{if } c = 0 \\ [x_0, x_{n-1}, \dots, x_1] & \text{if } c = 1 \\ \text{ror}(\text{ror}(x, c-1), 1) & \text{if } c > 1 \end{cases}$$

The bitwise **left rotate** function  $\text{rol} : \mathbb{F}_2^n, \mathbb{N} \rightarrow \mathbb{F}_2^n$  is defined as

$$\text{rol}(x, c) = \begin{cases} x & \text{if } c = 0 \\ [x_{n-2}, \dots, x_0, x_{n-1}] & \text{if } c = 1 \\ \text{rol}(\text{rol}(x, c-1), 1) & \text{if } c > 1 \end{cases}$$

Lastly, it is often practical to express numbers in a specific base. Hexadecimal numbers will be prefixed with 0x, such as 0xAB34. Binary numbers will be prefixed with 0b, such as 0b00101101. Decimal numbers will be written as-is, without prefix.

## Chapter 3

# Standardized interfaces and protocols

In order to comply with EU legislation laid out in Directive 98/69/EC of the European Parliament[8], each petrol car registered since January 1, 2001 and each diesel car registered since January 1, 2004<sup>1</sup> supports the EOBD (European On Board Diagnostics) standard. This mandates the use of the SAE J1962 connector (Figure 3.1a) and specifies pin assignments and protocols that should be available for diagnostic purposes. Differences between EOBD and OBD-II<sup>2</sup> standards are minimal, and may be ignored for the purpose of this thesis<sup>3</sup>. In this section, standards relevant to the work presented in the thesis are introduced.

### 3.1 The SAE J1962 connector

Commonly referred to as the OBD-II connector, The SAE J1962[25] specifies the connector socket that is legally required to be installed in abovementioned passenger vehicles. The female connector is present near the steering wheel of all recent passenger vehicles and is visible on Figure 3.1a, while the standardized part of the pinout is shown in Figure 3.1b. Pin 6 and 14 are used in conjunction with the CAN protocol, while communication over pin 7 and 15 uses the K-line protocol. Pin 2 and pin 10 are used for the J1850 serial interface and are shown for completion, but are not relevant for this thesis and will not be discussed.

---

<sup>1</sup>Passenger cars with no more than 8 seats and a gross weight of at most 2500 kg. Different dates apply for other categories and for newly introduced models, as can be found in Directive 98/69/EC.

<sup>2</sup>Compatibility with OBD-II is required for vehicles sold in the USA.

<sup>3</sup>Differences are discussed in [36]

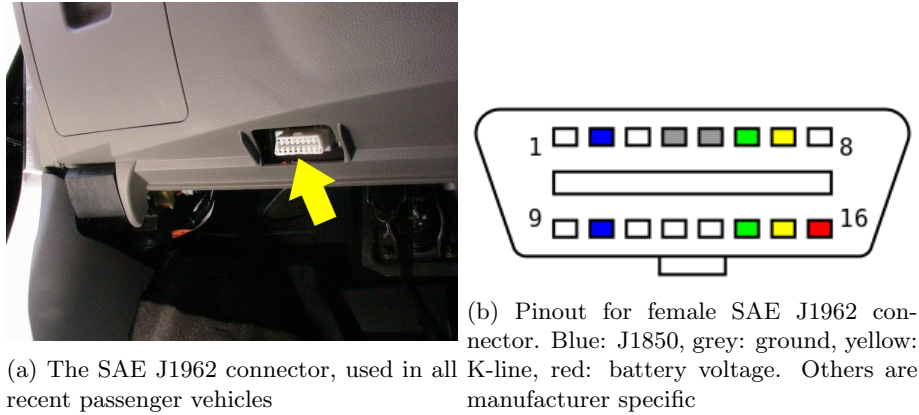


Figure 3.1: The SAE J1962 connector and the standardized pinout

### 3.2 The ISO 9141 K-line protocol

The K-line interface is standardized in ISO 9141[14], and uses Keyword Protocol 2000 (KWP2000)<sup>4</sup> for communication, as defined in ISO 14230[12]. K-line is a single wire interface<sup>5</sup>. K-line is a predecessor of CAN, and its use in passenger vehicles is declining. Since CAN is widely adopted in modern vehicles as the standard for both inter-component communication and diagnostics, and most of the work presented in this thesis is related to the CAN bus, the K-line interface will not be discussed in detail in this thesis.

### 3.3 The ISO 15765 CAN protocol

ISO 15765[13] defines Keyword Protocol 2000 (KWP2000) over CAN. It defines both the CAN frame format, how multi-frame messages are to be constructed and how diagnostic services are to be implemented. CAN is a two-wire interface, where transmission as a 0-bit is defined as actively driving the CAN HI wire to the high voltage<sup>6</sup> and the CAN LO wire to 0V. A 1-bit is transmitted by not driving either wire, resulting in CAN HI being 0V and CAN LO passively returning to a voltage by a resistor. This allows for elegant collisions resolution on the bus: if a sending party recognizes that a 0-bit is detected on the wire while it was itself sending a 1-bit, a collision is detected, and the detecting party will abort its transmission.

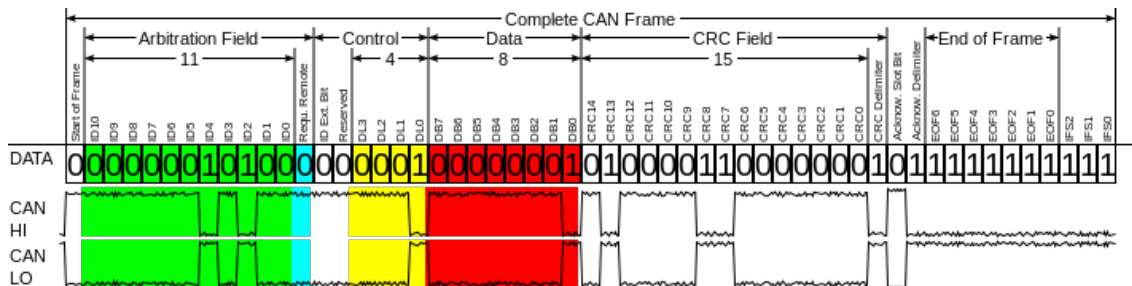


Figure 3.2: A CAN frame as it is transmitted over the bus. Source: Wikipedia

A CAN frame<sup>7</sup> starts with an 11 bit Arbitration Field, better known as the CAN ID, which can

<sup>4</sup>The same protocol is used in CAN, and KWP-2000 will be discussed in more detail in Section 3.3.

<sup>5</sup>An optional, second wire may be used, the L-line. However, in practice, it is rarely used.

<sup>6</sup>Which voltage is used for the high signal depends on the implementation

<sup>7</sup>We omit the notion of *stuffbits*, which are an addition for electro-technical reasons. Understanding the concept is not required for this thesis and would only add unnecessary complexity.

be interpreted as an identifier for the destination ECU. An ECU can listen for multiple IDs. Due to the nature of the collision resolving mentioned above, the CAN ID also acts as a bus arbitration mechanism. A 0-bit will have priority over a 1-bit, as any party simultaneously transmitting a 1-bit will detect the collision and abort its transmission. This implies that lower CAN IDs have priority over higher CAN IDs. The CAN ID is followed by seven control bits, among which four bits that define the number of bytes in the data field. The control bits are followed by 0 to 8 data bytes. Then, a 15-bit CRC checksum, followed by 10 end-of-frame bits, one of which allows the receiving party to acknowledge reception by sending a dominant 0-bit while the sending party sends a recessive 1-bit.

There also is an extended CAN frame definition. The main distinction is that extended frames use 29-bit CAN IDs instead of 11. More details may be found in the ISO specification.

In ISO 15765-2, the transport layer protocol is defined. This is commonly referred to as *ISO-TP*, and allows for the construction of both single-frame and multi-frame packets. Although multi-frame packets are very common, in this thesis, only single frame packets are encountered and as such, only the single frame format will be explained.

A CAN frame can contain up to 8 data bytes. The first byte is used by ISO-TP for frame type and size information. The first nibble designates frame type and will be 0 for a single-frame packet while the values 1, 2 and 3 are used for the construction of multi-frame packets. The second nibble designates the number of payload bytes. An ISO-TP frame can thus contain 7 payload bytes.

```
12:51:32.067 | can_id[7E0]: 0627069DBCE907]
```

Listing 3.1: An example of an ISO-TP message, as intercepted by a Jifeline tool.

In above example, a message is sent on CAN ID 0x7E0. The packet is a single frame (first nibble equals 0), with a six byte payload (second nibble) consisting of the bytes 0x27069DBCE907. Please note that not all CAN messages in this thesis are ISO-TP packets, as internal functionality often uses raw CAN data packets. The frame type and packet length nibbles are then omitted and all 8 bytes can be used as payload bytes.

### 3.4 The ISO 14229 UDS protocol

The UDS protocol is a diagnostic protocol, specified in ISO 14229[11]. Support is required by EU and US legislation. It defines a list of services that the vehicle must or could provide, and defines the request and response formats. The services that are relevant to this thesis are listed below, a full list can be obtained from the ISO standard documents<sup>8</sup>.

Identifier	Name	Description
0x10	Diagnostic Session Control	Opens a diagnostic session of some type.
0x11	ECU Reset	Resets the ECU. Different reset types exist.
0x22	Read Data By Identifier	Read a data element. This can be all kinds of information, such as sensor data, VIN, ECU number and more.
0x23	Read Memory By Address	Reads a part of the internal memory of the ECU. The address and the number of bytes to be returned are specified as parameters.
0x27	Security Access	Authentication in order to unlock security-critical services.

One service UDS defines is Diagnostic Session Control, which opens a session for various types of diagnostic functions. Often, authentication is needed before diagnostic procedures may be invoked, which is handled by the Security Access service. Authentication for Security Access is

<sup>8</sup>Softing published a poster summarizing all common services and response codes, which may act as a practical quick reference document. It can be found at [https://automotive.softing.com/fileadmin/sof-files/pdf/de/ae/poster/uds\\_info\\_poster\\_v2.pdf](https://automotive.softing.com/fileadmin/sof-files/pdf/de/ae/poster/uds_info_poster_v2.pdf)

done by means of a challenge-response. Upon receiving a Security Access request, the ECU sends a so-called *seed*, which serves as a challenge. The diagnostic device then authenticates by sending the correct response, referred to as (*key*). The cryptographic primitive for the challenge-response is chosen by the manufacturer. Many seedkey functions require authentication by means of the vehicle security code (or PIN).

12:20:27.038		can_id [18DA40F1]:	022707
12:20:27.068		can_id [18DAF140]:	066707BF795856
12:20:27.108		can_id [18DA40F1]:	0627080DDE3A63
12:20:27.137		can_id [18DAF140]:	026708

Listing 3.2: An example of a successful seedkey authentication, as captured by a Jifeline tool.

Listing 3.2 shows an example of how seedkey authentication may take place. The diagnostic device sends a seedkey request (0x27) of type 0x07. The type byte is always odd. The ECU responds with an acknowledgement (0x27 + 0x40 = 0x67) and sends the seed, 0xBF795856. The diagnostic device computes the correct response using the seed and possibly an additional secret, such as the vehicle security code, and sends the response (0x0DDE3A63), preceded by a 0x27 service byte and the type byte, incremented by one. After verification of the correctness of the response, the ECU then acknowledges the successful authentication.

## Chapter 4

# Literature review

The main focus of this thesis is on the assessment of cryptographic components of immobilizer systems. As such, this literature study focuses on research regarding vulnerabilities in modern vehicles and in the underlying cryptographic primitives. A fair amount of academic work has been done regarding secure ECU authentication and entity authentication in general. Some of this work will be presented in Section 10, as it can be used constructively in order to improve the security of ECM authentication.

In 2010, Kosher et al. pushished a paper titled "Experimental Security Analysis of a Modern Automobile" [27], in which they perform an in-depth study on one vehicle model. Using various techniques, they determine a large set of functional CAN packets, allowing for control over brakes, engine, door locks and more. The research does discuss access control on diagnostic functions and firmware flashing routines, and is further extended in a more formal way in Kosher's 2014 dissertation [26], however, ECM authentication is not discussed. Another detailed case study of a single vehicle was presented by Miller and Valisek in [30]. Focusing on remote access, they managed to find a remotely useable attack on a 2014 Jeep Cherokee. Using the remote IP of the vehicle, they were able to leverage unauthenticated remote access to the vehicle's D-Bus service in order to gain control over the CAN bus. Also, a large amount of scientific work has been presented on the subject of vehicle immobilization. This research however focuses on the transponder technology embedded in the car key, such as [43] and [42] by Verdult et al., on Megamos and Hitag2 respectively. The immobilizer component that authenticates the key, however, is generally the BCM, implying that another authentication must take place between the BCM and the ECM in order to prevent the engine from starting when an invalid key is inserted. Criminals are known to use tools that communicate over the CAN-bus [35], and as such, the security of this authentication is crucial in order to guarantee the safety of the immobilization system. During his presentation on Sigint13, K. Nohl stated [32] that theft of modern, immobilizer-equipped cars is not done by means of cryptographic attacks on immobilizer systems, but rather by attacking the other ECU functionality. While it is true that many ECUs implement weak authentication on functionality such as key programming or EEPROM access, this is only one way to circumvent security, and as such, strong ECM authentication is necessary to secure the vehicle against theft. The use of weak cryptographic components may also lead to false confidence, and as such, these components should be carefully assessed to ascertain the desired security guarantees can indeed be provided by the chosen protocols and cryptographic primitives.

It is noteworthy that knowledge about the internals of ECU authentication does seem to exist outside of the manufacturers, however, often only embedded in proprietary commercial tools. An example encountered during the research is the use of the SP Diagnostics tool for deriving the Model A immobilizer algorithm, as is discussed in Section 7.2. Abrites ltd [29]. is another company that has extensive knowledge on the internals of a wide array of vehicles. Their diagnostic devices support retrieval of vehicle security codes for a large amount of vehicles. Also, on various marketplaces, tools are available that bypass or disable immobilization without requirement for a valid key or security code. These products are highly suited to facilitate vehicle theft. An example

of such a tool is the JRL CAN adapter, which allows to program a new key for recent Range Rover vehicles while connected to the blind spot sensor, as shown in Figure 4.1.

### JLR Adapter to unlock the car and program key via CAN-BUS



JLR CAN Adapter is used for opening and programming keys to Jaguar, Land Rover and Range Rover Manufactured after 2010

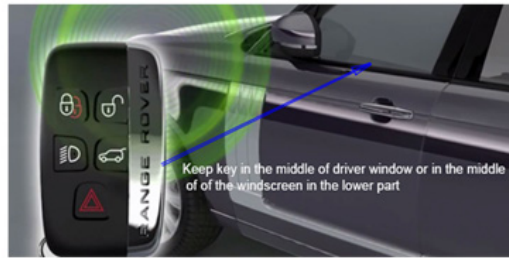
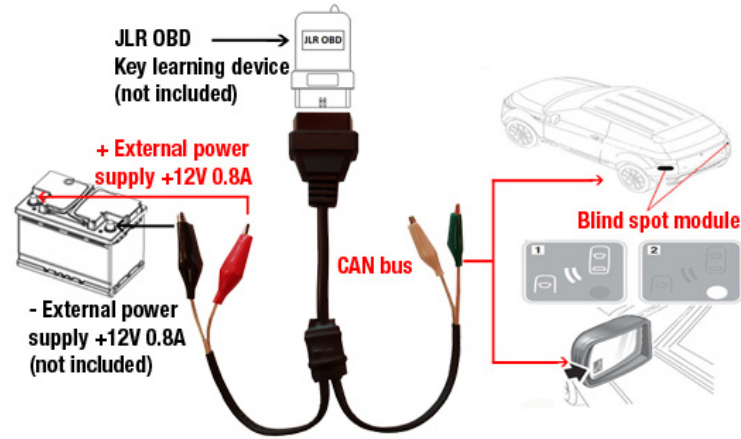


Figure 4.1: CAN adapter that allows for key learning while connected to the vehicle blind spot sensor. Source: [37]

## Chapter 5

# Methodology

In this section, the general methodology for extracting of the ECM authentication protocol from a vehicle will be introduced. While the exact approach differs for each model, there is sufficient overlap to justify a general overview, allowing us to be more concise when describing the process for each case study.



Figure 5.1: The Jifeline tool. It can be connected to a tablet using USB-OTG, which then forwards the CAN data over wifi. This allows for easy analysis as well as the possibility to interact and inject frames using Python.

Having chosen which vehicle to assess, I first observe the behavior of the vehicle on the CAN bus. Using a Jifeline (Figure 5.1), all messages on a CAN-bus are captured and stored for research. I investigate the messages that are sent when the key is inserted, and when the ignition switch is turned to the *ON* position. In general, one will observe large amounts of messages, while only a few will deal with the immobilizer system. However, using some simple heuristics, one can often easily identify the messages that *are* part of the immobilizer system. As we expect to observe a challenge-response, one such heuristic is to identify any message with high-entropy bytes that are different each time the vehicle is powered up<sup>1</sup>. If such a message is an actual challenge or response message, this implies a second message should have been sent slightly earlier or later<sup>2</sup>. Identification of two messages that fit this heuristic provides us with details about the presumed challenge/response message format.

<sup>1</sup>Often, the car has to remain switched off for about 10 seconds before such values are reinitialized.

<sup>2</sup>Generally, in the order of 100ms.

It is important to establish a high confidence in having found the actual challenge-response messages. This can be done in various ways. First, one can disconnect some components from the CAN-bus and observe changes in the traffic to learn which device sends which messages<sup>3</sup>. Another useful method is to inject challenges onto the bus, watching for a response from the other part of the immobilizer system. The response should be generated by a deterministic function depending on the injected challenge bytes. Lastly, it is interesting to remove the transponder from the keyfob and observe how this affects the immobilizer challenge/response.

Once the messages that are related to the immobilizer have been identified and the structure of the exchange has been analysed<sup>4</sup>, we have sufficient information to start investigating the firmware.

Obtaining the firmware can be done in various ways. Forums and other online resources exist where firmwares for many popular ECUs can be either downloaded freely or purchased. Another, possibly more reliable way, is to extract the firmware directly from an ECU. This way, one can be sure that the firmware dump corresponds to that particular ECU. As an additional advantage, it is often easier to know where in the memory space the firmware is loaded when it is extracted directly from the ECU. Lastly, one need not rely on the competence of some (often unfamiliar) individual who originally created the dump.

There are different approaches to dumping firmware directly from an ECU. One way is to use the UDS 0x23 ReadMemoryByAddress function. A successful 0x27 SecurityAccess (see Section 3.4) authentication must generally be done beforehand, which is possible if the seedkey algorithm and the vehicle security code are known. Another method is by leveraging the on-chip debugging functionality as provided by many modern microcontrollers. JTAG is commonly implemented and functional on ARM and MIPS-based microcontrollers<sup>5</sup>. BDM is a similar debugging interface, available on many Freescale microcontrollers based on ARM and PowerPC architectures. BDM and JTAG both provide a reverse engineer with powerful features, such as raw register/memory access and the usage of breakpoints. Using an on-chip debugging tool is a relatively easy way to obtain the firmware and/or analyse the behaviour of the firmware by debugging.

In order to start reverse engineering a firmware image, one needs to know some details about the microcontroller. Obtaining the microcontroller's datasheet, as well as an architecture reference document, is often crucial in the process of figuring out how the firmware image is loaded into memory and executed.

Once some details about the microcontroller are known, the firmware image can be loaded into a disassembler. For my research, I have been using Hex-Rays IDA Pro 6.95. IDA can disassemble binaries for a large variety of architectures, and while it is a static disassembler, it implements lots of tricks and heuristics that make a reverse engineer's life a lot easier. Scripting support makes it easy to automate some tedious tasks, allowing the user to load and execute Python-based scripts that make use of IDA's powerful API. Also, for some architectures, IDA is able to *decompile* the disassembly, effectively translating assembly into c-like pseudocode (Figure 5.2). While far from perfect, this often greatly helps in quickly understanding the behaviour of a function.

Once the firmware has been loaded into IDA at the right memory address<sup>6</sup>, segments can be created for RAM and I/O regions. As soon as these have been created at the appropriate addresses, IDA will recognize references to these areas. This makes it easy to find out which code or data regions contain references to some address under investigation.

Different strategies exist for finding the relevant cryptographic functions in a firmware dump. Searching for exclusive-or operations generally works well, as the operation is relatively rarely used in "regular" functions<sup>7</sup> while being popular in cryptographic functions. Also, cryptographic function usually exhibit a clearly recognizable structure. Little conditional branching (other than

---

<sup>3</sup>Naturally, this method is not fully accurate, as the messages may originate from an ECU that is still connected but expects a message from the disconnected ECU before sending some packet. Still, this approach often yields usable information.

<sup>4</sup>Useful features are for example the involved CAN IDs, opcodes, and the length of the non-static part.

<sup>5</sup>However, I did not personally investigate any ECU based on ARM or MIPS.

<sup>6</sup>For some microcontrollers, the flash memory where the firmware is stored is mapped to memory address 0, but this is certainly not always the case. The datasheet will provide the necessary details on where in memory the internal flash will be mapped.

<sup>7</sup>Asides from simple bitflip operations, which are easily recognized based on the constant that is being used.

```

seg000:00058C08 # void __fastcall memcpy(unsigned __int8 *dest, unsigned __int8 *source, unsigned int count)
seg000:00058C08 memcpy:                                     # CODE XREF: sub_2F038+260Tp ...
seg000:00058C08                                     # load_hword_from_tables+ECip ...
seg000:00058C08                                     srwi.    r6, r5, 2      # Shift Right Immediate
seg000:00058C0C                                     beq.     loc_58C28      # Branch if equal
seg000:00058C10                                     addi     r7, r3, -4     # Add Immediate
seg000:00058C14                                     addi     r8, r4, -4     # Add Immediate
seg000:00058C18                                     mtctr    r6            # Move to count register
seg000:00058C1C loop_copyWord:                         # CODE XREF: memcpy+1C4j
seg000:00058C1C                                     lwzu     r12, 4(r8)     # Load Word and Zero with Update
seg000:00058C20                                     stwu     r12, 4(r7)     # Store Word with Update
seg000:00058C24                                     bdnz     loop_copyWord # CTR--; branch if CTR non-zero
seg000:00058C28                                     #
seg000:00058C28 loc_58C28:                             # CODE XREF: memcpy+4fj
seg000:00058C28                                     clrrwi   r8, r5, 2     # Clear Right Immediate
seg000:00058C2C                                     cmplw    r8, r5        # Compare Logical Word
seg000:00058C30                                     bge      locret_58C58  # Branch if greater than or equal
seg000:00058C34                                     subf     r5, r8, r5     # Subtract from
seg000:00058C38                                     mtctr    r5           # Move to count register
seg000:00058C3C                                     add      r12, r3, r8    # Add
seg000:00058C40                                     add      r11, r4, r8    # Add
seg000:00058C44                                     addi     r7, r12, -1    # Add Immediate
seg000:00058C48                                     addi     r4, r11, -1    # Add Immediate
seg000:00058C4C                                     #
seg000:00058C4C loop_copyByte:                       # CODE XREF: memcpy+4C4j
seg000:00058C4C                                     lbzu     r12, 1(r4)     # Load Byte and Zero with Update
seg000:00058C50                                     stbu     r12, 1(r7)     # Store Byte with Update
seg000:00058C54                                     bdnz     loop_copyByte # CTR--; branch if CTR non-zero
seg000:00058C58                                     #
seg000:00058C58 locret_58C58:                         # CODE XREF: memcpy+28fj
seg000:00058C58                                     blr                                             # Branch unconditionally
seg000:00058C58 # End of function memcpy

```

(a) Disassembly

```

void __fastcall memcpy(unsigned __int8 *dest, unsigned __int8 *source, unsigned int count)
{
    unsigned __int8 *destPtr; // r702
    unsigned __int8 *srcPtr; // r802
    unsigned int numWords; // ctr02
    unsigned int remainingBytes; // r804
    unsigned int remainingCount; // ctr05
    unsigned __int8 *destPtr_2; // r705
    unsigned __int8 *srcPtr_2; // r805

    if ( count >> 2 )
    {
        destPtr = dest - 4;
        srcPtr = source - 4;
        numWords = count >> 2;
        do
        {
            srcPtr += 4;
            destPtr += 4;
            *destPtr = *srcPtr;
            --numWords;
        }
        while ( numWords );
    }
    remainingBytes = count & 0xFFFFFFFF;
    if ( (count & 0xFFFFFFFF) < count )
    {
        remainingCount = count - remainingBytes;
        destPtr_2 = &dest[remainingBytes - 1];
        srcPtr_2 = &source[remainingBytes - 1];
        do
        {
            **destPtr_2 = **srcPtr_2;
            --remainingCount;
        }
        while ( remainingCount );
    }
}

```

(b) Decompilation

Figure 5.2: A function, both disassembled (5.2a) and decompiled (5.2b) by IDA

based on counter values), no IO-related activity and few cross references (or *xrefs*) to both main and subfunctions (as they are typically only used in one or several places) are other features that are distinguishing for cryptographic functions in most firmware images.

## Chapter 6

# Assessed vehicles

We will briefly discuss the assessed vehicles. Due to a responsible disclosure process, the manufacturers and car models will not be published in this thesis, but will be made public at a later point in time.

The choice for the three models was motivated by availability of ECU test sets. The vehicles are listed in the order in which they were investigated. All ECM authentication protocols assessed in the course of this research are listed in this thesis. Research was performed on these models in the belief that, with high probability, the identified protocols would still be in use in currently produced vehicles. Verification learned this is indeed the case, as is detailed below.

### 6.1 Model A

The research on Model A was conducted on ECUs from a vehicle manufactured in 2009. Although this vehicle is no longer in production, we have confirmed the same ECM authentication protocol to be used in at least one model introduced in 2016, which as of today is still in production, and have confirmed the attack presented in 7.5.1 can still be carried out successfully.

### 6.2 Model B

The Model B ECU set originates from a vehicle constructed in 2009. While the model is no longer in production, we have confirmed the ECM protocol to be used in at least one vehicle that is still in production and was introduced in 2014. The attacks presented in 8.5 have been confirmed to work on this vehicle.

### 6.3 Model C

Our Model C ECU set was obtained from a vehicle built in 2008. However, the Model C ECM authentication protocol was confirmed to be present in the ECM of at least one vehicle introduced in 2015 and is still being manufactured.

## Chapter 7

# Case study: Model A

### 7.1 Identifying the protocol messages

When starting the assessment of model A, the CAN messages that related to the immobilizer functionality were easily identified by searching for high-entropy bytes that are different each time the ignition is switched on. Listing 7.1 shows a successful authentication between the BCM and the ECM.

The ECM initiates the communication by sending an opcode byte 0x00, followed by a 32-bit challenge. The BCM will, if an authorized key is present, send a response starting with opcode 0x04, followed by 32 response bits. The ECM will validate the response and if accepted, acknowledge this by sending a 0x02 opcode.

11:10:55.417		can_id[072]:	0051D70B2C	//	ECM CHALLENGE
11:10:55.458		can_id[0A8]:	04257670B6	//	BCM RESPONSE
11:10:55.421		can_id[072]:	0200000000	//	ECM ACK

Listing 7.1: Immobilizer message exchange

Using the Jifeline scripting API, I created a script that injects a challenge and obtains a response from the BCM. Using this, I injected related challenges and analysed the responses, which revealed a remarkable dependency between challenge bits and response bits, as is visualized in Listing 7.2 and will be discussed in more detail in Section 7.4.2.



## 7.2 Obtaining the algorithm

Although Listing 7.2 seems to suggest the algorithm has a simple structure, deriving the algorithm from a set of challenge/response pairs is extremely hard. My tutor pointed out that we had a proprietary tool in the office, based on an ARM microcontroller, that is able to derive the car PIN. This tool is manufactured by SP Diagnostics[6], and allows to derive the car PIN over the OBD-II connector, instructing the operator to repeatedly toggle between the ignition ON and OFF positions. In order to obtain the pin, the tool demands the ignition to be switched from off to on six times. It thus seemed highly probable that this tool captures the challenge/response pairs and derives the PIN based on a computation over these pairs. While the tool had JTAG headers on the PCB, halting the processor for on-chip debugging turned out to be impossible. Presumably, the halting function was disabled by the vendor. However, my tutor was aware of an embedded USB debug function, allowing to dump internal memory regions by address.

As I did not know where in the internal memory space the immobilizer algorithm is located, I decided the easiest way to find this is by dumping the stack at two distinct moments in time: once when the tool displays the main menu, and once when the tool is actually capturing challenges. My intuition was that the bottom part<sup>1</sup> would be identical, and a deviation would occur around the point where the model-specific immobilizer code is invoked. Indeed, a loader function was identified that loads an application from SD card into memory. Dumping the memory region where the application was loaded yielded the PIN derivation application code for this vehicle. Finding the code segments that actually dealt with computing the PIN was straightforward: cross references to easily identifiable strings (Figure 7.1) were present.

```

ROM:2C001409          DCB 0, 0, 0
ROM:2C00140C  aPinNotFound  DCB "PIN NOT FOUND",0 ; DATA XREF: pinRecovery+334↓o
ROM:2C00140C          ; ROM:off_2C038858↓o
ROM:2C0014EA          DCW 0
ROM:2C0014EC  aD_0        DCB "%d%",0 ; DATA XREF: pinRecovery+3B8↓o
ROM:2C0014EC          ; ROM:off_2C038860↓o
ROM:2C0014F1          DCB 0, 0, 0
ROM:2C0014F4  aPinCode    DCB "PIN Code:",0 ; DATA XREF: pinRecovery+44E↓o
ROM:2C0014F4          ; ROM:off_2C038868↓o ...
ROM:2C0014FE          DCW 0

```

Figure 7.1: Clearly recognizable strings were present in the binary. The data xrefs point to the location in memory where these strings are used.

A simple brute force algorithm was encountered, along with the authentication algorithm. A `candidatePin` variable is initialized to 0x30303030, which translates to 0000 in ASCII. The tool then checks if the pin satisfies all challenge/response pairs that were captured before, and increments<sup>2</sup> the PIN if not. The loop ends if either a valid `candidatePin` is found, or if the search space is exhausted, as is visible in Figure 7.2.

The information from the SP Diagnostics tool matches the code that executes the immobilizer authentication protocol which we recovered from an ECM firmware image. A decompilation of the transformation function as found in the ECM (as will be discussed in the next section) is shown in Figure 7.3.

The PIN can be any four-character combination of numbers and uppercase letters, with the exception of the I and O characters. This yields a total of  $34^4 = 1336336$  possible pin codes. The `computeResponse` function implements the algorithm that translates a PIN / challenge pair to a response, and will be discussed in the following section.

<sup>1</sup>As in, the higher memory addresses, thus, the oldest values on the stack.

<sup>2</sup>As in, selects the next valid PIN.

```

130 do
131 {
132     candidatePin = incrementPin(candidatePin);
133     if ( !candidatePin )
134     {
135         printToLCD("PIN NOT FOUND", 1, 0, 1);
136         return;
137     }
138
139     // For each challenge/response pair
140     for ( i = 0; i < 6u; ++i )
141     {
142         matchFound = 1;
143         targetResponse = responses[i];
144         if ( targetResponse != computeResponse(candidatePin, challenges[i]) )
145         {
146             // If some non-match is found, this pin candidate is wrong: abort
147             matchFound = 0;
148             break;
149         }
150     }
151 }
152 while ( !matchFound );
153
154
155
156
157
158
159
160
161
162
163

```

Figure 7.2: Brute force pseudocode snippet. Some lines were omitted for clarity.

```

1 int __fastcall T_1(int i, _WORD *dest)
2 {
3     int div; // r5@1
4     int rem; // r3@1
5     int diff; // r5@1
6
7     div = i / 0xB2;
8     rem = i % 0xB2;
9     diff = -63 * div + 0xAA * rem;
10    if ( (signed __int16)diff < 0 )
11        LOWORD(diff) = diff + 0x7673;
12    *dest = diff;
13    return rem;
14 }

```

Figure 7.3: Transformation function, as found in the decompiled ECM firmware.

### 7.3 Algorithm details

The immobilizer algorithm itself is based on a single transformation function  $T()$ . Although it is based on mathematical operands such as multiplication, division and modulo, it also relies on several low-level implementation details of C. In order to properly express the transformation function, we need to define a field containing all 32-bit signed integers. While the inputs and outputs of the transformation function are all 16-bit signed integers, defining the 32-bit signed integer field will remove the necessity of extensive overflow handling, improving the readability of the definition of  $T()$ .

**Definition 7.3.1.** The signed 32-bit integer field is defined as  $\mathbb{Z}_2^{32}$ , in two's complement bitwise representation.

The mapping of a 16-bit unsigned integer  $a \in \mathbb{F}_2^{16}$  to a 32-bit signed integer  $a' \in \mathbb{Z}_2^{32}$  is defined as follows

$$a' = \begin{cases} a & \text{if } a < 0x8000 \\ a \vee 0xFFFF0000 & \text{otherwise} \end{cases}$$

The mapping of a 32-bit signed integer  $a \in \mathbb{Z}_2^{32}$  to a 16-bit unsigned integer  $a' \in \mathbb{F}_2^{16}$  is defined as follows

$$a' = a \wedge 0xFFFF$$

We also need to define a division remainder operation, as the C modulo operator '%' does not behave according to the mathematical definition of modulo when a negative left-hand side operand is used.

**Definition 7.3.2.** The remainder  $rem : \mathbb{Z}_2^{32} \rightarrow \mathbb{Z}_2^{32}$  is defined as

$$rem(a, b) = a - (a/b) * b$$

**Definition 7.3.3.** The transformation function  $T(i, d, q, r) : (\mathbb{Z}_2^{32}, \mathbb{Z}_2^{32}, \mathbb{Z}_2^{32}, \mathbb{Z}_2^{32}) \rightarrow \mathbb{F}_2^{16}$  is defined as follows.

$$T(i, d, q, r) = \begin{cases} m - n & \text{if } m - n \geq 0 \\ m - n + a & \text{otherwise} \end{cases}$$

where  $m = rem(i, d) * r$ ,  
 $n = (i/d) * q$ ,  
 $a = d * r + q$

Note that the output of  $T()$  is cast to the  $\mathbb{F}_2^{16}$  field as defined in Definition 7.3.1. The transformation function is used by the main function, that splits the vehicle security code  $k$  and challenge  $c$  in parts and feeds them to the transformation function as parameter  $i$  in four distinct invocations.

**Definition 7.3.4.** The challenge/response function  $F(k, c) : (\mathbb{F}_2^{32}, \mathbb{F}_2^{32}) \rightarrow \mathbb{F}_2^{32}$  is defined as follows.

$$F(k, c) = ((a \vee b) \ll 16) \vee (c \vee d)$$

where

$$\begin{aligned} a &= (T((c[0] \ll 8) \vee c[2], 0xB2, 63, 0xAA, 0x7673)) \\ b &= (T((k[0] \ll 8) \vee k[3], 0xB1, 2, 0xAB, 0x763D)) \\ c &= (T((k[1] \ll 8) \vee k[2], 0xB2, 63, 0xAA, 0x7673)) \\ d &= (T((c[1] \ll 8) \vee c[3], 0xB1, 2, 0xAB, 0x763D)) \end{aligned}$$

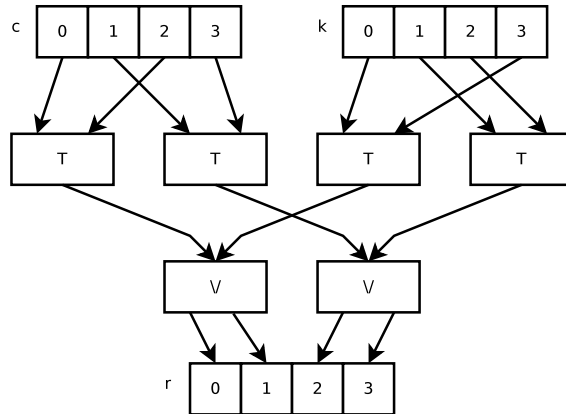


Figure 7.4: Illustration of the structure of the Model A algorithm. The parameters of  $T()$ , other than  $i$ , have been omitted.

## 7.4 Properties of the cipher

### 7.4.1 Insufficiently large keyspace

While even a 32-bit key would be insufficiently strong to protect against a brute-force attack, the key that is being used for the immobilizer system is the vehicle security code. For this model, as was stated before in Section 7.2, the vehicle security code consists of 4 characters, with 34 possibilities per character. The attack complexity for an exhaustive search would thus be  $34^4$ , which is approximately  $2^{20.35}$ .

### 7.4.2 Lack of diffusion

The challenge and the key are both split in two 2-byte parts. Due to the structure of the algorithm, the lower half of the response is only dependent on two challenge bytes and two key bytes. The upper half of the response is only dependent on the remaining two challenge bytes and two key bytes. This lack of interdependency makes that the algorithm does not adhere to the strict avalanche principle, as defined by Webster et al. in 1986<sup>3</sup>. This is clearly visible in the traces shown in Listing 7.2.

### 7.4.3 Inverse of the transformation function

The transformation function requires four parameters. The first one,  $i$ , is a 16-bit value<sup>4</sup> derived from either the challenge or the PIN, while the remaining three are fixed constants. As the transformation function outputs a 16-bit value<sup>5</sup>, it acts as a mapping from a 16-bit integer to a 16-bit integer.

Due to the nature of the transformation function, multiple values for  $i$  will result in the same output. As such, the inverse function  $T^{-1}$  is not bijective. However, as the domain of the PIN is limited (Section 7.4.1), only few pre-images could possibly have been derived from the vehicle security code. Indeed, in all cases, there is only one unique pre-image for  $T()$  if we know the pre-image has been derived from part of the PIN.

<sup>3</sup>Webster et al. defined the *strict* avalanche principle, which is an extended definition of the avalanche principle defined by Feistel[10] in 1973. Webster et al. introduce a desired probability of  $p = 0.5$  of any output bit to flip if any single input bit were flipped. Indeed, the Model A algorithm does not even adhere to the weaker original definition of the avalanche principle, as each output bits is fully independent of a large amount of input bits.

<sup>4</sup>Cast from  $\mathbb{F}_2^{16}$  to  $\mathbb{Z}_2^{32}$

<sup>5</sup>Cast back from  $\mathbb{Z}_2^{32}$  to  $\mathbb{F}_2^{16}$

#### 7.4.4 Biased responses

Due to the use of the logical **or** operator to merge two outputs from the transformation function ( $a \vee b$  and  $c \vee d$  in Definition 7.3.4), a bias is introduced in the output. Assuming the output of the transformation function is uniform random, we can expect 75% of the bits in the output domain of  $F()$  to be 1, and thus, on average 75% of the bits in the response will be set<sup>6</sup>. The average amount of observed bias is dependent on the PIN, as a PIN that will result in low hamming weight output from  $T()$  will exhibit bias to a lesser extent.

#### 7.4.5 Leakage of key information

The cause of the previously mentioned bias also causes another undesired property. One can abuse the bitwise **or** operation to derive the output of  $T()$  for the two parts of the PIN  $k$ . This can be done by simply observing several responses. Bits that are always set are also set in the corresponding output of  $T()$ , while bits that vary are not set. This can clearly be seen in Listing 7.2. As the output values of  $T()$  with respect to  $k$  have been derived, a list of pre-images can be constructed, yielding several key candidates.

**Theorem 7.4.1.** *Given response  $r$  and transformation function output  $a$  and  $d$ . It then holds that*

$$b_i = \begin{cases} 0 & \text{if } a_i = 0 \wedge rl_i = 0 \\ 1 & \text{if } a_i = 0 \wedge rl_i = 1 \\ \text{undefined} & \text{otherwise} \end{cases} \quad \forall i \in (0, 15)$$

$$c_i = \begin{cases} 0 & \text{if } d_i = 0 \wedge rr_i = 0 \\ 1 & \text{if } d_i = 0 \wedge rr_i = 1 \\ \text{undefined} & \text{otherwise} \end{cases} \quad \forall i \in (0, 15)$$

where  $a, b, c, d$  are the outputs of  $T()$  as defined in Definition 7.3.4,  $rl = r \gg 16$ ,  $rr = r \wedge 0xFFFF$ .

Thus, we can obtain on average 1/2 of the bits of  $b$  and  $c$ , depending on the hamming weight of  $a$  and  $d$ . If we obtain the response corresponding to  $c = 0x00000000$ , both  $a$  and  $d$  equal zero, resulting in full recovery of  $b$  and  $c$ .

### 7.5 Attacks

For this model, I devised two attacks that derive the car security code. Both are detailed below.

#### 7.5.1 Attack with valid key

This attack will retrieve the vehicle security code. A key with an authorized transponder must be present in the ignition lock.

As was stated in 7.4.1, the car PIN is only four characters long, with 34 possibilities per character. Clearly, the resulting key space is insufficient to prevent an attacker from mounting a successful brute force attack based on a set of intercepted challenge/response pairs. My C implementation of a brute force search exhausts the search space in approximately 0.3 seconds<sup>7</sup>. The number of challenge/response pairs to obtain a single pin satisfying all pairs is variable, but when random pairs are captured, five pairs is usually sufficient.

However, one can also inject chosen challenges onto the CAN-bus and capture the response. Following the property given in Section 7.4.5, capturing the response associated with  $c = 0x00000000$

---

<sup>6</sup>This bias is not visible in Listing 7.2, as the challenges contain a large amount of zeros. This leads to small output values of the transformation function, explaining the relatively low amount of 1 bits in the output.

<sup>7</sup>Single threaded, on an Intel i5-4200U CPU.

will result in exactly one candidate PIN. Deriving the car security code using this attack takes less than one second. This attack was verified to work against multiple other models from the same concern<sup>8</sup>.

### A more efficient variant

A far more efficient attack is possible. Combining the properties found in Sections 7.4.5 and 7.4.3, we can identify the outputs of  $T()$  that were based on the vehicle security code (referred to as  $b$  and  $c$ ), then find the corresponding pre-image. The PIN is easily derived, as only one pre-image exists. The required outputs of  $T()$  can either be obtained by observing multiple challenge/response pairs or by injecting challenge 0x00000000 and analyzing the response, as this single challenge will result in full recovery of  $b$  and  $c$ .

## 7.5.2 Car-only attack

Contrary to the previous attack, this attack does not require a valid key to be present. It will allow an attacker to either deactivate the immobilizer system (effectively allowing the car to start), or obtain the vehicle security code.

A characteristic of the algorithm is that given a challenge  $c$ , there are many PIN codes  $k$  that will result in the same response  $r$ . This is due to the use of the bitwise **or** operation for combination of values derived from  $k$  and  $c$ , as pointed out in Section 7.4.5. Due to the small keyspace, we can easily compute the responses for all PINs given a challenge  $p$ , then grouping those by response. We define the set  $K_r$  as the set containing all PINs  $k$  that yield response  $r$  for a given challenge  $c$  and candidate PIN set  $K$ , as defined formally below.

**Definition 7.5.1.** For a given challenge  $c$  and set of candidate PINs  $K$ , the set  $K_r$  is defined as

$$k \in K_r \iff k \in K \wedge F(c, k) = r$$

This can be used to construct a car-only attack. The BCM will provide the attacker with a challenge. When an incorrect response is sent to the BCM, it immediately sends a new challenge. No rate limiting is implemented, and authentication can be attempted at a rate of up to 5 times per second.

The attack works as follows. Initially, all PINs are potentially correct, so  $K$  will contain each  $k$ . When presented with a challenge  $c$ , the set  $K_r$  is constructed for this challenge by computing the associated response for each  $k \in K$ . Now pick response  $r$  that corresponds to the largest set  $K_r$ . By attempting to authenticate with this response, we have a small chance to authenticate successfully, with  $p = |K_r| / |K|$ . If the authentication fails, we have found that each pin in  $K_r$  is invalid. We can now update  $K$  by taking  $K \leftarrow K \setminus K_r$ . Having removed the PINs from the candidate pin set  $K$ , we now repeat the process with a new ECM challenge.

Simulations have shown that on average 4000 challenge/response attempts are needed in order to successfully authenticate, resulting in an expected optimal attack time of approximately 15 minutes.

---

<sup>8</sup>Although this algorithm is used by multiple car models, some small implementation differences were encountered. For some models, injection of challenges is less reliable, as the ECM also continuously sends challenges to the BCM, even when the engine is running. The decrease in reliability was solved by simply intercepting the ECM challenges and associated BCM responses, and computing the car PIN as soon as five pairs have been acquired. This is slightly slower, the attack takes about 10 seconds.

## Chapter 8

# Case study: Model B

### 8.1 Identifying the protocol messages

For this model, the challenge/response messages were again easily identified by searching for messages containing variable high-entropy parts upon switching on the ignition. A trace of a successful authentication is shown in Listing 8.1.

14:05:29.459		can_id[0010A001]:	0508938F220E53
14:05:29.461		can_id[0010A000]:	AB1FDE
14:05:29.474		can_id[0010A001]:	06

Listing 8.1: Message exchange of successful authentication

The ECM initiates the challenge/response by sending opcode 0x05, followed by six bytes with variable values. The BCM replies with opcode 0xAB, followed by two variable bytes. The ECM acknowledges a correct response with an opcode 0x06. My initial intuition was that the challenge consisted of six bytes, expecting a two-byte response. However, when trying to obtain responses for chosen (injected) challenges, the BCM responds with an error code 0x10 (Listing 8.2).

14:05:40.102	can_id[0010A001]:	0512345678ABCD
14:05:40.104	can_id[0010A000]:	10000000000000

Listing 8.2: Message exchange with injected challenge

The reason for this behaviour was found in the firmware, as will be discussed in the following section.

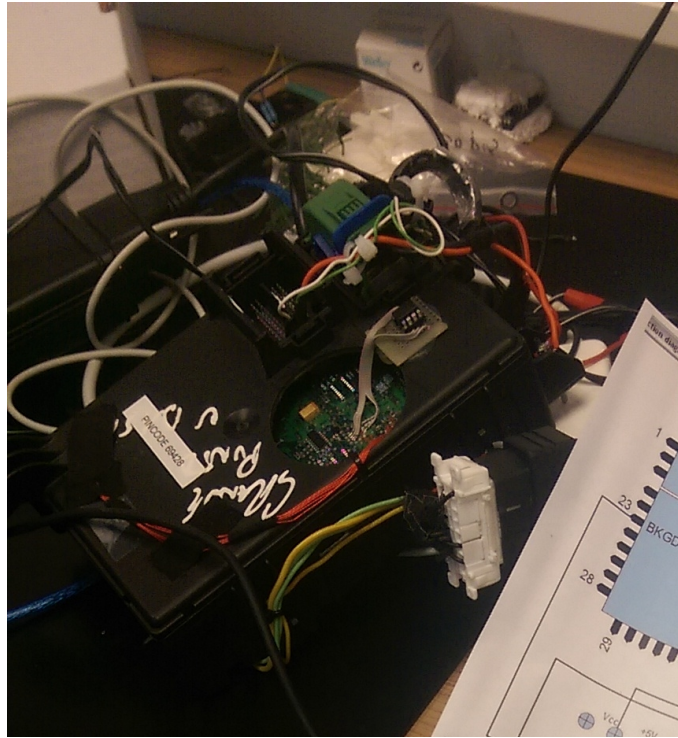


Figure 8.1: The BCM from Model B.

## 8.2 Obtaining the algorithm

I obtained a firmware image from the BCM, which was based on the NEC<sup>1</sup> V850ES<sup>2</sup> architecture. This is a 32-bit RISC architecture with some particularities when compared to architectures like x86, PowerPC and ARM. For instance, it features two hardware registers that have to be initialized to point to certain base addresses in memory: the GP (Global Pointer) register and the CTBP (CALLT base pointer) register. Serving as a base address for relative references to RAM and a table of commonly used functions<sup>3</sup>, initialization to the correct address is required in order to be able to properly follow the control flow. A search for write accesses to these registers yielded a few candidate values, while further analysis allowed for identification of the one combination that yielded sensible results across the firmware.

Despite my best efforts, at first, I was unable to locate the immobilizer-related cryptographic functions in the BCM. Therefore, I started working on a dump of an ECM from the same vehicle, based on a 16-bit STMicroelectronics ST10F280 microcontroller<sup>4</sup>. In this firmware, the cryptographic function responsible for the challenge/response authentication was identified by analysing functions that use exclusive or instructions. Having found the algorithm, I created a first C implementation based on the code extracted from ST10F280 firmware. Then, presence of the algorithm in the BCM was verified. As the algorithm makes use of a 32-bit constant that is absorbed into the state upon initialization, a search for this constant allowed for straightforward confirmation of the algorithm's presence in the BCM firmware. This allowed for easier reverse engineering, as 32-bit V850ES assembly has a very high information density when compared to the ST10F280's 16-bit architecture.

A second implementation was made, based on the 32-bit assembly extracted from the BCM.

<sup>1</sup>After merging with Renesas in 2010, sold under the Renesas brand name.

<sup>2</sup>Datasheet available online: [http://pdf.datasheet.company/datasheets-1/reneasas\\_electronics/UPD70F3237M1GJ\\_A\\_-UEN.pdf](http://pdf.datasheet.company/datasheets-1/reneasas_electronics/UPD70F3237M1GJ_A_-UEN.pdf)

<sup>3</sup>In this firmware, mainly different kinds of frequently used function prologues / epilogues.

<sup>4</sup>Datasheet available online: [http://www.keil.com/dd/docs/datashts/st/st10f280\\_ds.pdf](http://www.keil.com/dd/docs/datashts/st/st10f280_ds.pdf)

I thoroughly inspected the code and created a third version, that more elegantly implements the algorithm for Intel x86 / AMD64 architectures instead of merely translating the V850/ES assembly.

The algorithm uses three input parameters. Two 32-bit values are used to initialize the internal state of the algorithm. One 16-bit value is loaded after approximately half of the rounds of the algorithm have been performed. Also, at separate phases of the algorithm, a 12-bit and a 14-bit output value are constructed. When comparing this behaviour with traces of successful authentications, such as in Listing 8.1, the format of the challenge response packets became clear. The ECU sends a 0x05 opcode, followed by a 32-bit challenge, followed by the 12-bit output. This was visible, because in all traces, the most significant four bits of the last two byte part are zero. A similar phenomenon was visible in the response: after the 0xAB opcode, the first nibble of the two following bytes was always a number between 0 and 3: clearly, this is where the 14-bit output is used. Concluding, the 14-bit output is the actual response, while the 12-bit output serves as a proof of knowledge of the secret, serving to convince the BCM that the challenge was indeed generated by the ECU.

Naturally, I had to verify the correctness of my implementations. Renesas CS+, a development environment provided by Renesas, is equipped with an emulator for V850ES based targets. Loading the firmware, setting GP and CTBP to their respective values, and setting breakpoints and PC to the desired locations allowed for emulating parts of the firmware, which was used to generate sample traces for validation purposes.

### 8.3 Algorithm details

The algorithm is constructed around a 32-bit Fibonacci LFSR<sup>5</sup> and consists of four stages. First, the internal state is initialized, using a 32-bit secret  $k$ . Then, 12 bits of output  $p$  (used as a proof of knowledge on  $k$ ) are generated. This is followed by loading an additional 16-bit secret  $l$  into the internal state, followed by the generation of the actual response  $r$ . The four phases are discussed in detail in the following sections, while the high-level definition of the algorithm is given here.

---

#### Code Listing 1 Model B authentication algorithm

---

```
function MODELB_AUTH( $c, k, l$ )
   $st \leftarrow P1(c, k)$ 
   $st, p \leftarrow P2(st)$ 
   $st \leftarrow P3(st, l)$ 
   $st, r \leftarrow P4(st)$ 
  return  $p, r$ 
```

---

#### 8.3.1 Phase 1: Initialization

During the first stage, the 32-bit state register  $st$  is initialized based on the 32-bit secret  $k$ , challenge  $c$  and a constant:

$$st \leftarrow k \oplus c \oplus 0x0E080004$$

After setting the initial value, the round function is invoked 38 times. Each round, eight state bits are used to construct a selector byte  $s$ , which is used to perform a table lookup from the feedback table (Appendix A.1) and generate a new state bit. The selector byte  $s$  is derived from the state  $st$  as defined in Definition 8.3.1.

---

<sup>5</sup>LFSRs, or linear feedback shift registers, are a commonly used, light-weight solution to generating keystream based on a secret initial state. More information on LFSR types and theory can be found in various resources online, such as [24] and [5].

**Definition 8.3.1.** The selector byte  $s$  is defined as

$$s = [st_{31}, st_{14}, st_{21}, st_{12}, st_{19}, st_{26}, st_1]$$

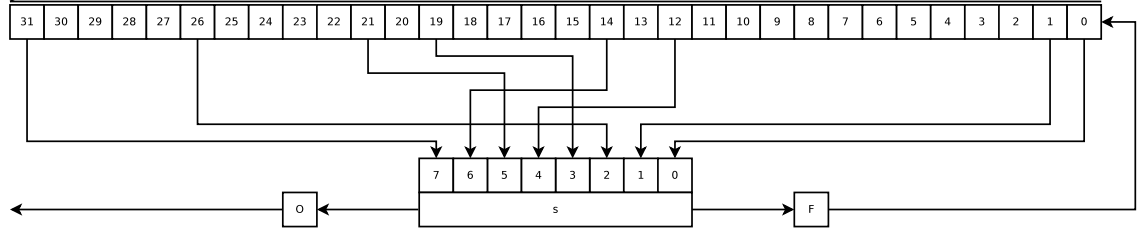


Figure 8.2: Construction of selector  $s$  from  $st$ . The feedback bit is generated by function  $F(s)$ . Phase 2 output bits are generated by function  $O(s)$ .

Each round, a new state bit is generated, and the state is shifted one position to the left. The generated bit becomes the least significant state bit  $st_0$ . In order to generate  $st_0$ , the selector byte  $s$  is split into the upper 5 bits, to be used as an index, and the lower three bits, to be used as a mask.

**Definition 8.3.2.** The feedback function  $F(st) : \mathbb{F}_2^{32} \rightarrow \mathbb{F}_2$  is defined as

$$F(st) = \text{FeedbackTable}[s_{[s_7, \dots, s_3]}]_{7-[s_2, \dots, s_0]}$$

where  $s$  is derived from  $st$  as defined in Definition 8.3.1.

**Definition 8.3.3.** The round function  $R(st) : \mathbb{F}_2^{32} \rightarrow \mathbb{F}_2^{32}$  is defined as

$$R(st) = st \ll 1 + F(st)$$

For example, if  $s = 0x42 = 0b01000010$ , we have  $[s_7, \dots, s_3] = 0b01000$  and  $[s_2, \dots, s_0] = 0b010$ . The feedback table byte at index 8 is  $0x66$  or  $0b01100110$ . We select bit  $7 - [s_2, \dots, s_0] = 5$ , which is set. Thus, a state that yields  $s = 0x42$  would generate a 1 as feedback bit. Although a byte is obtained from the feedback table, the construction using the mask has the advantage of requiring less storage space.

The LFSR, the construction of selector  $s$  and the feedback function are illustrated in Figure 8.2. Pseudocode of the phase 1 initialization function is given below.

---

**Code Listing 2** Phase 1

---

```

function P1( $c, k$ )
     $st \leftarrow k \oplus c \oplus 0x0E080004$ 
    for  $i = 0$  to 37 do
         $st \leftarrow R(st)$ 
    return  $st$ 
    
```

---

### 8.3.2 Phase 2: Proof generation

After the state has been initialized, 12 bits of output are generated. These 12 bits are sent along with the challenge to the BCM, and serve to prove that the challenge was generated by someone who knows secret  $k$ , asserting that only the ECM can send a valid challenge/proof pair. We will refer to the generated proof as  $p$ .

The generation of the proof is done by generating one output bit after each invocation of the round function. This is done in a similar way as the feedback bit generation, using the same selector byte  $s$  but a different lookup table, given in Appendix A.2.

**Definition 8.3.4.** The output function  $O(st) : \mathbb{F}_2^{32} \rightarrow \mathbb{F}_2$  is defined as

$$O(st) = \text{OutputTable}[s_{[s_7, \dots, s_3]}]_{7-[s_2, \dots, s_0]}$$

where  $s$  is derived from  $st$  as defined in Definition 8.3.1.

Twelve output bits are generated, from left to right. That is, after the first invocation of the round function  $p_{11}$  is generated, followed by  $p_{10}$  after the second round, and so on until the least significant bit  $p_0$  has been generated. The pseudocode for phase 2 is given below.

---

**Code Listing 3** Phase 2

---

```

function P2( $st$ )
   $p \leftarrow 0$ 
  for  $i = 0$  to 11 do
     $st \leftarrow R(st)$ 
     $p_{11-i} \leftarrow O(st)$ 
  return  $st, p$ 

```

---

### 8.3.3 Phase 3: Second secret absorption

After proof  $p$  has been generated, a value derived from the 16-bit secret  $l$  is absorbed into the state. We will refer to the transformed value as  $m$ , the transformation  $T()$  is defined below.

**Definition 8.3.5.** The function  $T(l) : \mathbb{F}_2^{16} \rightarrow \mathbb{F}_2^{32}$  is defined as

$$T(l) = (l \oplus 0x11) \ll 6$$

Note that the left shift occurring in  $T()$  implies the lower six bits of  $m$  are zero. Also, as the first bit of  $m$  that is used in  $P3()$  is  $m_{19}$ ,  $l_{15}$  and  $l_{14}$  are never used and do not affect the state. The pseudocode for phase 3 is given below.

---

**Code Listing 4** Phase 3

---

```

function P3( $st, l$ )
   $m \leftarrow T(l)$ 
  for  $i = 0$  to 19 do
     $st \leftarrow R(st)$ 
     $st_0 \leftarrow st_0 \oplus m_{19-i}$ 
  return  $st$ 

```

---

### 8.3.4 Phase 4: Response generation

The fourth phase is structured identically as the proof generation phase. The response  $r$  is generated by invoking the round function 14 times. After each invocation, output function  $O(st)$  is used to compute the next response bit. As is the case in the proof generation phase, the most significant response bit is generated first. The pseudocode for the generation of response  $r$  is given below.

**Code Listing 5** Phase 4

---

```

function P4( $st$ )
   $r \leftarrow 0$ 
  for  $i = 0$  to 13 do
     $st \leftarrow R(st)$ 
     $p_{13-i} \leftarrow O(st)$ 
  return  $st, r$ 

```

---

## 8.4 Properties of the cipher

### 8.4.1 The feedback function

The algorithm is shaped as a 32-bit Fibonacci LFSR with 8 taps for the feedback function  $F()$ . As was mentioned in Section 8.3.1, the feedback function uses a selector byte  $s$  to generate a single feedback bit using the feedback table in Appendix A.1. However, analysis of the feedback table reveals it can be rewritten as a linear operation on  $s$  that uses only four state bits.

$$F(s) = s_7 \oplus s_5 \oplus s_1 \oplus s_0$$

This can be mapped directly to the state bits that were used to construct  $s$ .

**Lemma 8.4.1.** *The feedback function  $F(st): \mathbb{F}_2^{32} \rightarrow \mathbb{F}_2^{32}$ , as defined in Definition 8.3.2, is equivalent to*

$$F(st) = st_{31} \oplus st_{21} \oplus st_1 \oplus st_0$$

The resulting simplified feedback LFSR is visualized in Figure 8.3.

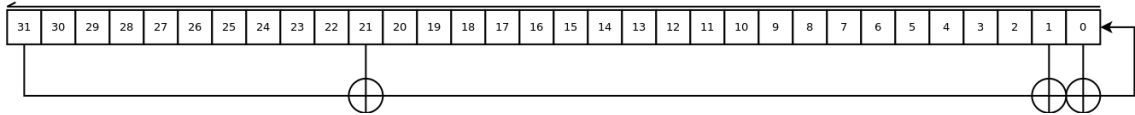


Figure 8.3: The feedback LFSR with four taps.

It is interesting to note that although the table-based implementation uses five table index bits ( $[s_7, \dots, s_3]$ ) and three output bit selection bits ( $[s_2, \dots, s_0]$ ), only four bits are of influence in the feedback function  $F(st)$ .

### 8.4.2 Inverse round function

As the round function operates as an LFSR, once the state is known, it can be rolled back easily. Although eight state bits are used to select the feedback bit, we have shown in Section 8.4.1 that only four state bits are used for generating the feedback bit. As the feedback function is a linear operation, an inverse function can be found.

**Lemma 8.4.2.** *The inverse round function  $R^{-1}(st): \mathbb{F}_2^{32} \rightarrow \mathbb{F}_2^{32}$  is given by*

$$R^{-1}(st) = ((st_{22} \oplus st_2 \oplus st_1 \oplus st_0) \ll 31) + (st \gg 1)$$

### 8.4.3 Knowledge of state bits based on output bit

If at some point in time, an output bit<sup>6</sup> is generated, we know it is some bit from a byte in the output table (Appendix A.2). Which bit is selected is determined based on the selector byte  $s$ , which consists of eight state bits. Given an output bit, there are exactly 128 values for  $s$  that would have resulted in the observed output bit. Assume  $s_g$  is a correctly guessed value of  $s$ . We can now reconstruct 8 bits of the internal state, namely, the state bits that were used to construct  $s$ .

## 8.5 Attacks

Having derived the algorithm, I developed three incrementally efficient attacks. All these attacks are car-only: they do not require an authorized transponder key to be present. This is possible due to the fact that the ECM sends a proof in conjunction with the challenge, which allows for reconstruction of the 32-bit secret  $k$ . Using these attacks, the actual response  $r$  cannot be computed, since we lack the ability to infer the second secret  $l$ , which is 16 bits in size.

The inability to derive  $l$  is irrelevant from an adversary's perspective. This is due to the fact that the vehicle security code can be derived from  $k$ . The procedure to derive the security code is as follows: Consider  $k$  written down as a hexadecimal number. Take the five rightmost nibbles. For each nibble, if the value is greater than 9, subtract 9. If the resulting value is 0, set it to 7. The remaining decimal-digit-only hexadecimal string is the vehicle security code.

An example will clarify this procedure. Suppose we have  $k = 0xB15C03F2$ . Omitting the rightmost three nibbles yields  $0xC03F2$ . We subtract 9 from all non-decimal nibbles, yielding  $0x30372$ . We have one 0-nibble, which we replace by 7, resulting in a car security code of 37372.

This method of deriving the vehicle security code from the master secret  $k$  has its drawbacks. Clearly, the keyspace is only  $9^5 = 59049$ , which means an exhaustive search on the PIN would be of a complexity of approximately  $2^{15.85}$ . Additionally, the PINs are biased, as hexadecimal digits  $0xA \dots 0xF$  map to the decimal digits  $1 \dots 7$ , implying 8 and 9 are less common<sup>7</sup>. This could be used to prioritize certain guesses when mounting a brute force attack, allowing an adversary to, on average, find the correct PIN before having tried half of the search space. However, rate limiting mechanisms are in place, alleviating the risk of a successful brute force on seedkey authentication.

Deriving the value of  $k$  is thus an interesting venue of attack, as it would allow us to obtain the vehicle security code using the method outlined before. The PIN, derived from  $k$ , can then be used to authenticate a diagnostic session and, for instance, program a new key to be authorized by the vehicle. This would allow an adversary to successfully deactivate the immobilizer without any need to obtain  $l$ .

### 8.5.1 Naive exhaustive-search

The first attack consists of an exhaustive search on  $k$ , similar to the one presented in 7.5.1. However, as the immobilizer algorithm employs a 32-bits key  $k$ , the search space is a lot larger than was the case for the Model A attack. The attack works by trying all possible values of  $k$ , and comparing whether or not the key will match a known set of challenge / proof pairs. Single-threaded on an Intel i5-4200U, the key space is exhausted in 166 minutes.

### 8.5.2 Pruning exhaustive-search

The naive exhaustive search computes the entire response for each candidate key, before checking whether or not the computed response matches the observed response. A significant improvement in running time may be obtained by performing this check bit-by-bit as soon as a proof bit is generated. This way, when an incorrect key is used, the algorithm has a 50% chance *each round*

---

<sup>6</sup>Either a proof bit or a response bit.

<sup>7</sup>Assuming the 32-bit secret  $k$  is uniformly distributed across cars.

to detect a difference between the generated and the observed proof bits. As soon as a difference is detected, the candidate key can be disregarded. This yields an average of two rounds of proof generation required to detect an invalid key. Instead of running for 50 rounds, the algorithm will now run for 40 rounds on average.

However, further optimization is possible by omitting the 38 initialization rounds, and finding a post-initialization state (the state after the initialization phase) that will generate the observed proof. Once identified, this state can be rewound by 38 rounds in order to obtain the corresponding initial state. The inverse round function is given in Section 8.4.2.

The key  $k$  can be found by computing the exclusive or of the initial state, the challenge and the 0x0E080004 LFSR initialization constant. The average number of rounds required to detect an invalid key is now reduced from 50 to 2. The running time of the attack was reduced to approximately 90 seconds<sup>8</sup>

### 8.5.3 State reconstruction

Although the previous attack already yielded a fairly low attack time, a more efficient attack is possible. The generated proof bits reveal information about the internal state, as they are generated based on the value of selector  $s$ . For each generated output bit, 256 possible values of  $s$  must be considered. Half of those can be discarded right away, as  $O(st)^9$  must yield the observed output bit. This leaves us with 128 candidate values for  $s$ . As the state determines the value of  $s$  (Figure 8.2), the opposite also holds: a candidate selector value determines eight bits of the associated candidate state. This is visualized in Figure 8.4.

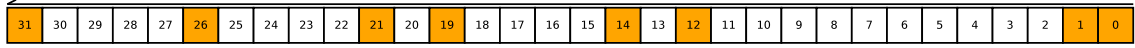


Figure 8.4: The orange state bits can be inferred from a given selector  $s$ .

This can be extended to a multi-round guess-and-determine attack. We start out at round 0 and observe the first generated proof bit  $p_{11}$ . There are 128 values for  $s$  that generate the observed proof bit  $p_{11}$ . We can now apply the round function on each partially known candidate state.

This depth in the recursive algorithm will correspond to round 1 of the proof generation step. Once more, we generate a list of candidate selectors, of which 128 will generate the observed proof bit  $p_{10}$  for round 1. However, we have less options to consider for  $s$  than in the previous round. The value of  $st_1$  does not need to be guessed, as we have already assumed its value in the previous round:  $st_0$  was shifted to the left and now determines  $st_1$ . Similarly, we already have the new value of  $st_0$ , as we have generated it by applying the round function on the previously assumed state. On average, there are thus only 32 candidate values for  $s$ .

In the following rounds, more and more state information is inferred. The state bits that need to be guessed each round are visualized in Figure 8.5. Using this principle, we can construct a recursive algorithm that generates a list of all keys  $k$  that, combined with the observed challenge  $c$ , would result in the observed proof  $p$ . To help building this algorithm, we define a *knowledge mask* for each round, which indicates for a round  $n$  which bits are assumed known (marked by a 1 in the knowledge mask), and which bits are free from assumptions. This helps to detect conflicts in assumed knowledge and candidate selectors, allowing to skip invalid selector candidate values.

**Definition 8.5.1.** The knowledge mask  $M_n$  at the start of round  $n$  is defined as

$$M_n = \begin{cases} 0 & \text{if } n = 0 \\ \text{rol}(M_{n-1} \vee 0x84285003) & \text{otherwise} \end{cases}$$

<sup>8</sup>This is faster than the expected factor of 25, which is explained by several other tweaks that improved performance.

<sup>9</sup>Note  $s$  is derived from  $st$ , according to Definition 8.3.4.

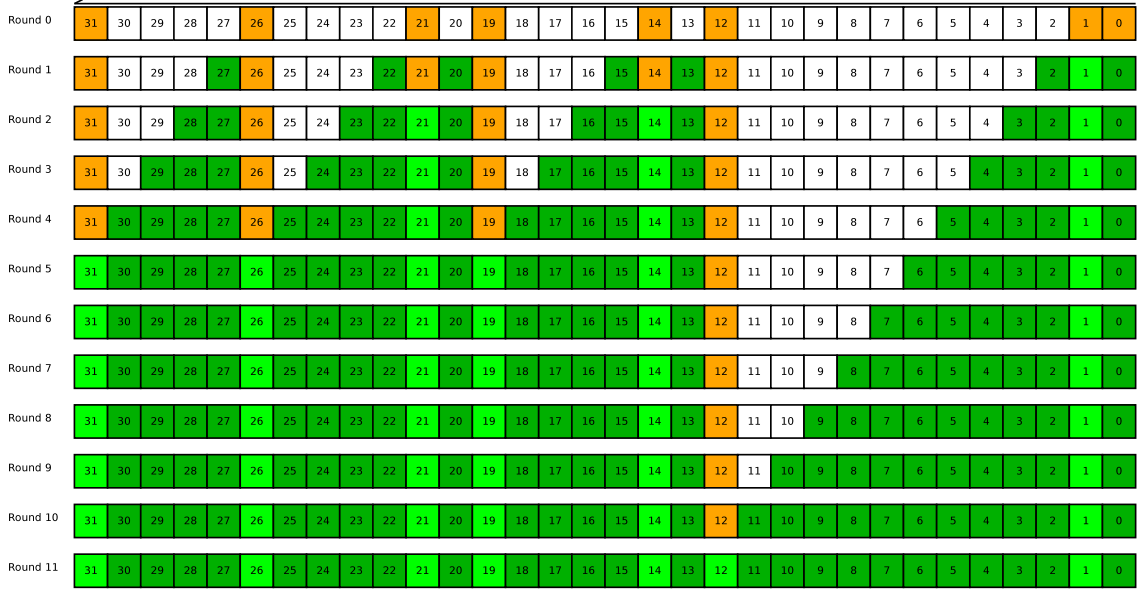


Figure 8.5: Propagation of knowledge about state bits per round. Orange bits have to be guessed. Dark green bits are known as a guess from previous rounds. Light green bits are selector bits that are fixed by previous guesses.

We can now define our recursive algorithm `RecursiveSearch()`, along with a helper function `RollbackToKey()`, which will be discussed shortly.

---

**Code Listing 6** Recursive search algorithm
 

---

```

function RECURSIVESHARE( $c, p, st, n$ )
    if  $n = 12$  then
        print RollbackToKey( $c, st$ )
    for  $s = 0$  to  $0xFF$  do
         $st' \leftarrow S(s)$ 
        if  $(p_{11-n} = O(st'))$  then
            if  $st \wedge M_n = st' \wedge M_n$  then
                 $st' = R(st \vee st')$ 
                RecursiveSearch( $c, p, st', n + 1$ )
    
```

---

Each time `RecursiveSearch()` is called with depth  $n = 12$ , we have found a valid LFSR sequence, that would have resulted in the observed proof  $p$ . The corresponding key is then easily derived by using the inverse round function to roll back to the initial state of the algorithm and then extracting the key from the initial state. This is done by `RollbackToKey()`, as is defined in Code Listing 7.

`RecursiveSearch()`, initially called with the challenge and proof while supplying zero values for  $st$  and  $n$ , thus lists all keys  $k$  that would have resulted in the observed proof  $p$ .

When testing against a sample challenge/proof pair, we see in Figure 8.6 that the number of invocations grows exponentially up to round 5 (which is reached 2134283 times), where it stabilizes. At the start of round 11, the inferred state information is complete (in other words, our knowledge mask is  $0xFFFFFFFF$ ), and as such, no further branching occurs. Instead, half of these states will generate a conflicting last output bit  $r_0$ , which explains why depth 12 is reached half as often as depth 11. We can now test each identified candidate key against other captured

**Code Listing 7** Rollback function

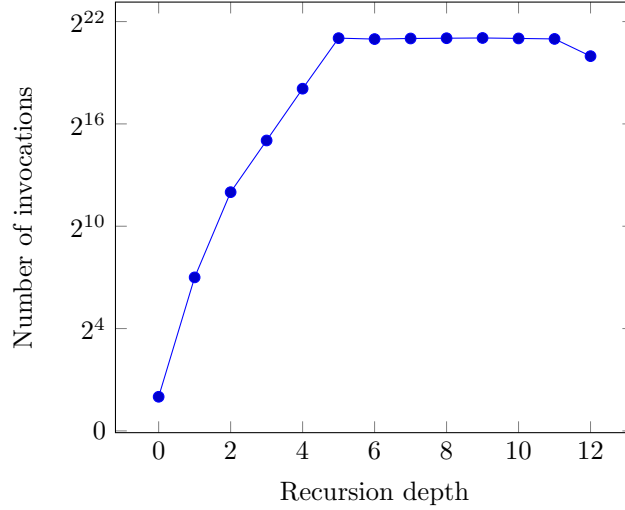
---

```

function ROLLBACKTOKEY( $c, st$ )
  for  $i = 0$  to 49 do
     $st \leftarrow R^{-1}(st)$ 
  return  $st \oplus c \oplus 0x0E080004$ 

```

---

Figure 8.6: Total number of `RecursiveSearch()` function calls per round depth.

challenge/response traces, after which the true key  $k$  is easily identified. The attack complexity is approximately  $2^{21}$  and a single-threaded run on an Intel i5-4200U CPU takes approximately 8 seconds.

#### 8.5.4 Deriving the second secret $l$

The 16-bit secret  $l$  that is required to compute the actual response  $r$  can also be recovered using above attack methods. However, in order to mount such an attack, an adversary needs to obtain a set of valid challenge/response pairs. In order to capture these pairs, the BCM needs to acknowledge the presence of an authorized transponder key in the ignition lock. Surprisingly, deriving vehicle security code can be done without presence of an authorized transponder key, as cooperation from the BCM is not required. Based on the recovered 32-bit key  $k$ , the vehicle security code can be computed, which can in turn be used to program a new key. This key will now be accepted and the immobilizer can be deactivated without the need to recover the value of  $l$ .

## Chapter 9

# Case study: Model C

### 9.1 Identifying the protocol messages

My research on Model C started with an analysis of CAN traffic on a test set. This test set comprises of an ECM, BCM and IPC module. I found that when the ignition is switched on, different "challenge/response-like" messages can be identified.

12:40:31.379		can_id[040]:	0023FAE51E8B
12:40:31.388		can_id[050]:	01024A740000
12:40:31.392		can_id[048]:	01014A740000
12:40:31.394		can_id[068]:	01204A740000
12:40:31.576		can_id[040]:	0000FAE50E80
12:40:34.156		can_id[040]:	0023FAE51E8B
12:40:34.172		can_id[050]:	01024A740000
12:40:34.173		can_id[048]:	01014A740000
12:40:34.181		can_id[068]:	01204A740000
12:40:34.355		can_id[040]:	0000FAE50E80
12:40:34.524		can_id[040]:	0023FAE51E8B
12:40:34.536		can_id[050]:	01024A740000
12:40:34.536		can_id[068]:	01204A740000
12:40:34.544		can_id[048]:	01014A740000
12:40:34.729		can_id[040]:	0000FAE50E80
12:40:34.922		can_id[040]:	0000FAE50680
12:40:35.113		can_id[480]:	101104A5D7CEFC68 // ECM challenge
12:40:35.147		can_id[488]:	0766020000000000 // BCM empty response
12:40:35.212		can_id[480]:	101104A5D7CEFC68 // ECM challenge
12:40:35.229		can_id[488]:	0766420059BA58EE // BCM response

Listing 9.1: Model C challenge/response messages

These messages can be separated in two independent authentication mechanisms. One is initiated by CAN id 0x040, while the other is initiated by 0x480. We will refer to these mechanisms as the 0x040 authentication and 0x480 authentication respectively, both will be discussed in the following sections.

#### 0x040 authentication

We see that CAN id 0x040 repeatedly sends a message, to which different components in the 0x040-0x068 range respond. This is part of a system that verifies that multiple components (ECM, BCM, IPC<sup>1</sup> and ACM<sup>2</sup>) in the car all have knowledge of the vehicle security code. This is done by means of a challenge, that varies per vehicle but is static each time the authentication takes place, followed by responses from three devices, once again diversified per vehicle without

---

<sup>1</sup>Instrument Panel Cluster

<sup>2</sup>Airbag Control Module

freshness guarantees per authentication. Also, note the responses from each authenticating device are equal (in this case, 0x4A74).

Despite the obvious weaknesses in this authentication system, it does not play a crucial role in the security of the vehicle. If this authentication fails, the car will not be immobilized. Instead, the only observable consequence is that the immobilizer status indicator on the IPC starts blinking, indicating to the user that there is a component authentication problem<sup>3</sup>. Due to the fact that this protocol does not play a role in the immobilization system, no further research has been done to uncover the internals of this challenge response algorithm.

### 0x480 authentication

The 0x040 authentication is followed by the actual immobilizer authentication step, initiated by a challenge sent on CAN id 0x480. The first two bytes are an opcode, 0x1011. This is followed by two status bytes, with value 0x04A5 in Listing 9.1. The exact value may vary by a few bits, and although the exact significance of each bit is not clear, it is not part of the challenge. The challenge consists of the last four bytes, and are different each time the authentication takes place. The first time the challenge is sent, a response message is sent with all-zero response bytes and status bytes 0x0200<sup>4</sup>. The challenge is repeated, followed by the actual BCM response, along with status bytes 0x4200. No acknowledgement message is sent, instead, the BCM ceases to send the challenge message<sup>5</sup>.

In contrast to the 0x040 authentication messages, the 0x480 authentication does relate to the actual immobilizer functionality: failure will lead to the ECM refusing to start the engine, and the immobilizer status indicator will light up continuously.

## 9.2 Obtaining the algorithm

In order to obtain the algorithm behind the immobilizer protocol, I started reverse engineering several firmwares of different cars from the same manufacturer.

In order to do more thorough research and verify that the found algorithm was actually used as a challenge-response algorithm, I obtained a test set consisting of only an ECM and a BCM from the same model. The ECM is manufactured by Bosch and equipped with a Freescale MPC555 microcontroller, based on a 32-bit PowerPC architecture. Freescale microcontrollers support BDM, which is a standard for on-chip debugging, allowing one to halt processor execution. In the halted state, one can inspect and update CPU registers and memory<sup>6</sup>. Also, a hardware breakpoint can be set, halting execution as soon as the breakpoint address is reached. This allowed me to verify that the found algorithm was actually executed, and obtain the key and challenge loaded on each run.

---

<sup>3</sup>According to the manufacturers, this kind of authentication system is intended to decrease the potential profit from trading stolen car parts. However, it also serves as an additional barrier when replacing parts with legitimate second-hand components. In most cases, a garage is able to reprogram the car security code in the donor part in order to successfully authenticate during this phase.

<sup>4</sup>The reason for this behavior is unknown. Possibly, the BCM has not yet had time to validate the transponder key.

<sup>5</sup>If no valid key is present, the ECM will send the challenge message 19 times, then stop transmitting the message. Immobilization will still be active.

<sup>6</sup>With memory, we mean the full internal 32-bit address space. Of course, not all addresses are readable or writeable, this depends on whether flash, I/O or RAM is mapped to the requested address.



Figure 9.1: The ECM from model C, with the Multilink FX on-chip debugging tool connected to the BDM debug pads.

### 9.3 Algorithm origin

The obtained algorithm turned out to be closely related to an immobilizer transponder family designed by Philips: PCF7935. This is a passive LF transponder introduced around 1994<sup>7</sup>, designed to be fully backward compatible with two related, but more light-weight transponder types, the PCF7930 and the PCF7931. Several security issues in these transponder types were addressed with the PCF7935, which implements a challenge/response algorithm based on a 128-bit symmetric secret in conjunction with, according to the datasheet, 48-bit challenges in order to generate an equally long response. The PCF7935 was superseded by the PCF7936 transponder type, better known as HITAG2[19].

Interestingly, Model C uses a key with a HITAG 2 passive LF transponder. Internally, the BCM emulates the behaviour of a PCF7935 security transponder in order to authenticate itself towards the ECM. A simplified message sequence chart of this authentication scheme is given in Figure 9.2.

We were surprised to find the BCM emulates an obsolete transponder for internal authentication, and sought to find a rationale behind this scheme. Our hypothesis was that the use of an

---

<sup>7</sup>I was unable to find any press statement or other clear evidence of a launch date of the PCF7935 transponder. However, in 1994, the first cars were introduced that use the PCF7935 transponders.

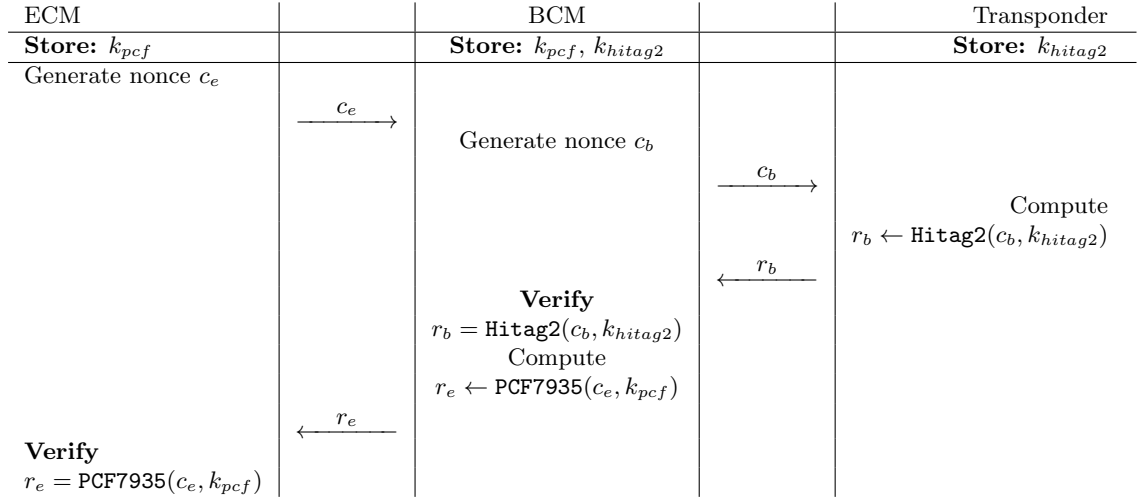


Figure 9.2: Model C situation: BCM and transponder authenticate using Hitag 2, while BCM and ECM authenticate using PCF7935.

”emulated” PCF7935 transponder was due to legacy requirements. It is possibly hard to upgrade the firmware of an ECM to a new transponder technology, since engines are developed separately from vehicles. If a new vehicle is equipped with a newer transponder technology, the need for emulation of the old, obsolete transponder becomes a necessity in order to be able to authenticate towards the ECM.

This was confirmed by investigating an older model from the same manufacturer. This vehicle was equipped with a PCF7935 transponder in the ignition key, and indeed, we found the ECM authenticates directly with the security transponder. An immobilizer box that serves as the LF interface only checks if the world-readable transponder identifier matches an authorized key. If so, it forwards the transponder response to the ECM. A simplified message sequence chart of this design is given in Figure 9.3.

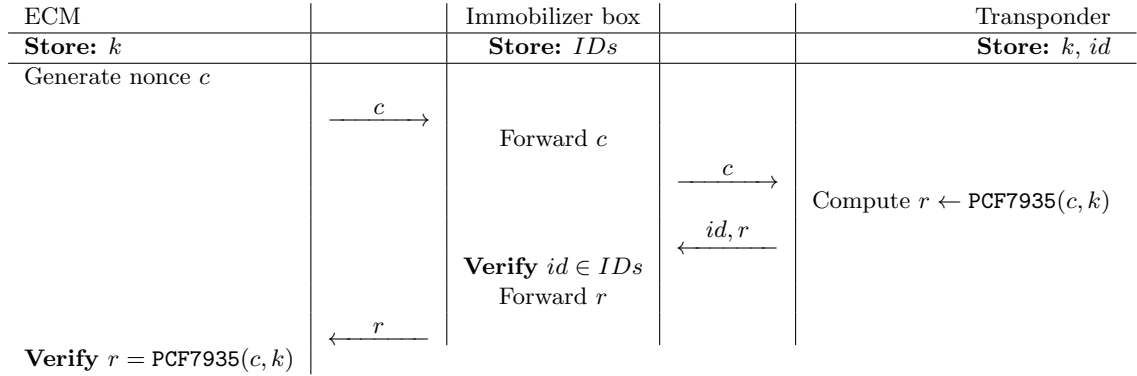


Figure 9.3: Old situation: immobilizer box forwards communication between ECM and transponder, and only verifies if transponder id is known.

When designing a first-generation immobilizer system, this seems a logical approach. However, the aforementioned compatibility issues may arise, which possibly pushed manufacturers to abandon this model in favour of the current approach: the BCM validates the key, after which the BCM authenticates towards the ECU if a valid key is present. This approach, however, introduces additional surface for an attack, as two instead of one challenge/response mechanism are in place.

As we are dealing with a software implementation of a physical LF transponder, it is relevant to have some understanding of the transponder design. The PCF7935 is a passive RFID transponder, operating on 125KHz. Access control flags allow for the restriction of write access to its 768 bits of user memory. It also incorporates a challenge/response algorithm, which uses a public 32-bit identifier (the IDE) and a 128-bit symmetric secret (referred to as **shadow** bytes) in order to generate a 48-bit response for a 48-bit challenge. A more detailed overview can be found in the PCF7935 datasheet, which can be found online<sup>8</sup>.

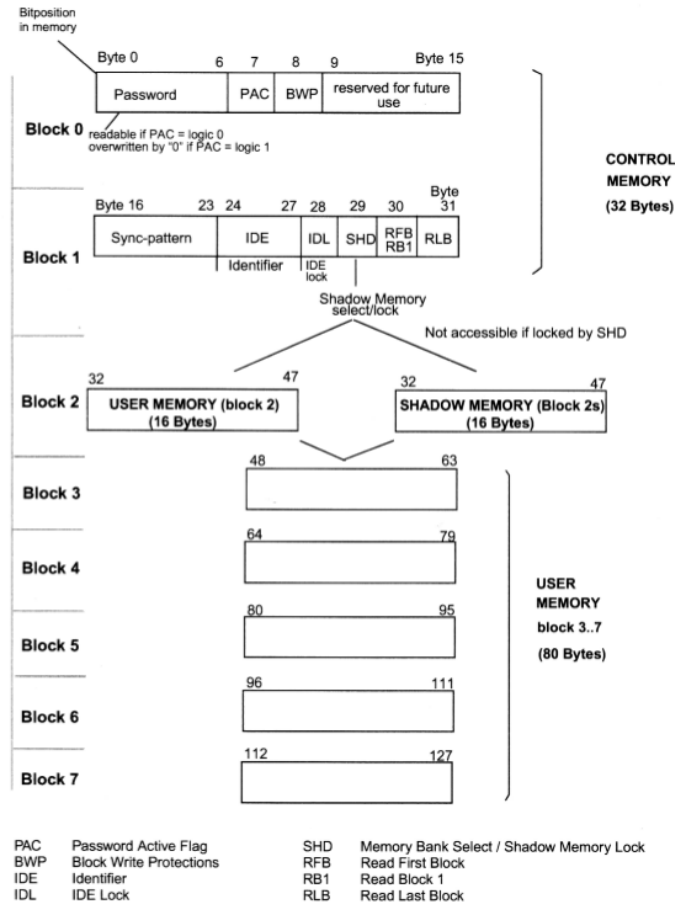


Figure 9.4: The PCF7935 internal memory layout. Source: NXP

As illustration 9.4 shows, the memory of the PCF7935 consists of 8 blocks of 16 bytes. Block 0 and block 1 are control memory, while block 2 to 7 are user memory. Some control registers are there for backwards compatibility with the older PCF7931 and PCF7930 transponder types, such as the 56-bit password. Of interest are the IDE field, which holds a four-byte identifier that is specific per car manufacturer<sup>9</sup>, and the SHD byte, which determines whether or not the shadow memory is accessible. When a transponder is personalized, the vehicle will set SHD byte such that accesses to memory block 2 are redirected to memory block 2s: the shadow memory. The vehicle then writes a 128-bit symmetric key to shadow memory and then sets the lock bits such that the shadow memory is no longer accessible and the symmetric key cannot be retrieved. The contents

<sup>8</sup>Datasheet: <http://datasheet.datasheetarchive.com/originals/distributors/Datasheets-309/70124.pdf>, memory layout: [http://www.computersolutions.cn/blog/wp-content/uploads/2011/02/pcf7930\\_35.pdf](http://www.computersolutions.cn/blog/wp-content/uploads/2011/02/pcf7930_35.pdf)

<sup>9</sup>For instance, a fictive car manufacturer "Carman" could choose to use an IDE value of 0x4341524D, which translates to the ascii string "CARM". Of course, a manufacturer could also choose to diversify the value of the IDE per vehicle.

of the shadow memory block serve as the secret symmetric key in the PCF7935 challenge-response algorithm, and are referred to as the *shadow bytes*.

## 9.4 Algorithm details

The algorithm uses a 48-bit challenge  $c$ , a 32-bit manufacturer identifier  $IDE$ , and a 128-bit key  $s$  in order to generate a 48-bit response  $r$ .

The authentication algorithm can be divided in five phases. During the first phase, the state and key byte array are initialized. The second phase absorbs 32 challenge bits into the state, while the third phase generates 32 response bits, by means of a very similar round function. The fourth phase is identical in structure to the second phase, and absorbs an additional 16 challenge bits. This is followed by the fifth and final phase that generates 16 more response bits. The high level algorithm is presented below, while each phase will be detailed in the following sections.

---

### Code Listing 8 Model C authentication algorithm

---

```

function MODEL_C_AUTH( $c, s, IDE$ )
     $st, k, kbp \leftarrow P1(c, s, IDE)$ 
     $st, kbp \leftarrow P2(st, c, k, kbp)$ 
     $st, kbp, r \leftarrow P3(st, k, kbp)$ 
     $st, kbp \leftarrow P4(st, c, k, kbp)$ 
     $st, kbp, r \leftarrow P5(st, r, k, kbp)$ 
    return  $r$ 
    
```

---

#### 9.4.1 Phase 1: Initialization

Upon initialization, the 32-bit state  $st$  and the 29-byte key array  $k$  are initialized with the aforementioned 128-bit *shadow bytes*  $s$  and the 32-bit  $IDE$ , as is illustrated in Figure 9.5. Pseudocode of this phase is given in Code Listing 9. Note that  $k[0 : 11] = k[16 : 27]$ , and  $k[12] = k[28]$ . A key byte pointer,  $kbp$ , is initialised to zero, and will be used as an incrementing pointer (modulo 29) to a byte in key array  $k$ .

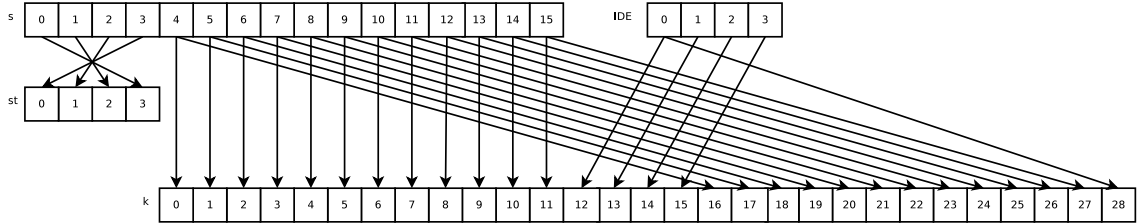


Figure 9.5: Initialisation of  $k$  and  $st$  from shadow bytes  $s$  and identifier  $IDE$ .

#### 9.4.2 Phase 2: Partial challenge absorption

During this phase, 32 bits of the challenge  $c$  are absorbed into the state. In order to absorb a single bit of the challenge into the internal state, the round function is invoked 17 times. During phase 2, the round function is thus invoked 544 times in total. The absorption phase feedback function  $F_A()$  and the absorption phase round function  $R_A()$  functions are defined below, followed by pseudocode for  $P2()$  which is given in Code Listing 10.

**Definition 9.4.1.** The absorption round function  $R_A(st, c_i, k, kbp) : \mathbb{F}_2^{32}, \mathbb{F}_2, \mathbb{F}_2^{29}, \mathbb{F}_2^{128} \rightarrow \mathbb{F}_2^{32}$  is defined as follows.

$$R_A(st, c_i, k, kbp) = (st \gg 8) + ((ror(st[3]) \oplus F_A(st, c_i) \oplus k[kbp]) \ll 24)$$

---

**Code Listing 9** Phase 1
 

---

```

function P1( $c, s, IDE$ )
    for  $i = 0$  to 3 do
         $st[i] \leftarrow s[3 - i]$ 
         $k[i + 12] \leftarrow IDE[i]$ 
    for  $i = 0$  to 11 do
         $k[i] \leftarrow s[i + 4]$ 
         $k[i + 16] \leftarrow s[i + 4]$ 
     $k[28] \leftarrow k[12]$ 
     $kbp \leftarrow 0$ 
    return  $st, k, kbp$ 
    
```

---

with  $F_A()$  as given by Definition 9.4.3.

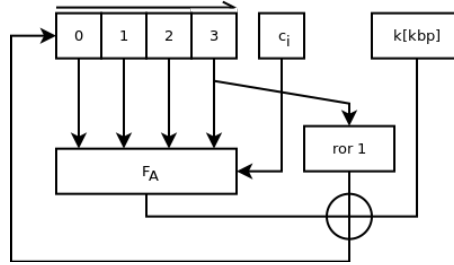


Figure 9.6: Phase 2: Challenge absorption round function  $R_A(st, c_i, k, kbp.)$

The round function computes a new state byte as the **xor** of three bytes. The first byte is a right-rotate by one bit of state byte  $st[3]$ . Second is the output of a nonlinear absorption phase feedback function  $F_A()$  over state and challenge bit. Lastly, a byte from the key array  $k$  is used. The state is shifted to the right by one byte, whereafter  $st[0]$  is set to be the newly computed state byte. This process is illustrated in Figure 9.6.

The absorption phase nonlinear feedback function  $F_a()$  is given by Definition 9.4.3. The challenge bit and state bits are used to compose four selector bytes  $s_a, s_b, s_c, s_d$ . The first selector  $s_a$  is then used as index for a table lookup in the Model C feedback table, as given in Appendix B.1. The retrieved byte  $a$  is masked to retain only two bits, that will eventually be part of the returned byte. The remaining three selectors are used in a similar way, however, each one is altered depending on the value of a prior table lookup. Eventually, four lookups have been performed, and the values are merged by addition<sup>10</sup>, yielding the result.

**Definition 9.4.2.** The absorption phase selectors are defined as

$$\begin{aligned}
 s_a &= [st_{22}, st_{30}, 0, st_{12}, st_{11}, st_{26}, st_{17} \vee c_i, st_8] \\
 s_b &= [st_{31}, st_{14}, st_{13} \oplus \overline{c_i}, st_{28}, st_{19}, st_{18}, st_9, 0] \\
 s_c &= [st_{23}, st_6, st_{29}, st_4, 0, 0, st_{25}, st_{16}] \\
 s_d &= [st_{10}, 0, st_{21}, st_{20}, st_{27}, st_2, st_1, st_{24}]
 \end{aligned}$$

where  $c_i$  is the challenge bit as supplied to the feedback function  $F_A()$ .

---

<sup>10</sup>As the union of the masks equals 0xFF, and the intersection of the masks equals 0, the bits in  $a, b, c$  and  $d$  do not overlap.

**Definition 9.4.3.** The absorption phase nonlinear feedback function  $F_A(st, c_i) : \mathbb{F}_2^{32}, \mathbb{F}_2 \rightarrow \mathbb{F}_2^8$  is defined as follows.

$$\begin{aligned}
 F_A(st, c_i) &= a + b + c + d \\
 \text{where} \\
 a &= \text{LookupTable}[s_a] \wedge 0xA0 \\
 b &= \text{LookupTable}[s_b \vee a_7] \wedge 0x09 \\
 c &= \text{LookupTable}[s_c \vee (b_0 \ll 2)] \wedge 0x12 \\
 d &= \text{LookupTable}[s_d \vee (c_1 \ll 5)] \wedge 0x44
 \end{aligned}$$

with  $s_a, s_b, s_c, s_d$  as given by Definition 9.4.2, and LookupTable as given by Appendix B.1.

Phase 2 only absorbs challenge bits  $c_{47}$  down to  $c_{16}$ . The remaining part of the challenge is absorbed in phase 4. Pseudocode of phase 2 is given in Code Listing 10.

---

**Code Listing 10** Phase 2

---

```

function P2( $st, c, k, kbp$ )
  for  $i = 0$  to 31 do
    for  $j = 0$  to 16 do
       $st \leftarrow R_A(st, c_{47-i}, k, kbp)$ 
       $kbp \leftarrow kbp + 1 \bmod 29$ 
  return  $st, kbp$ 

```

---

### 9.4.3 Phase 3: Partial response generation

The round function during response generation is nearly identical to the round function in phase 2. One difference is in the fact that no challenge bit is absorbed. The output phase round function  $R_O()$  and the corresponding nonlinear feedback function  $F_O()$  and selector definitions are given below. The round function  $R_O()$  is visualized in Figure 9.7.

**Definition 9.4.4.** The output generation round function  $R_O(st, k, kbp) : \mathbb{F}_2^{32}, \mathbb{F}_2^{29}, \mathbb{F}_2^{128} \rightarrow \mathbb{F}_2^{32}$  is defined as follows.

$$R_O(st, k, kbp) = (st \gg 8) + ((st[3] \oplus F_O(st) \oplus k[kbp]) \ll 24)$$

with  $F_O()$  as given by Definition 9.4.6.

**Definition 9.4.5.** The output phase selectors are defined as

$$\begin{aligned}
 s_a &= [st_{22}, st_{30}, 0, st_{12}, st_{11}, st_{26}, st_{17}, st_8] \\
 s_b &= [st_{31}, st_{14}, st_{28}, st_{19}, st_{18}, st_9, 0] \\
 s_c &= [st_{23}, st_6, st_{29}, st_4, 0, 0, st_{25}, st_{16}] \\
 s_d &= [st_{10}, 0, st_{21}, st_{20}, st_{27}, st_2, st_1, st_{24}]
 \end{aligned}$$

where  $c_i$  is the challenge bit as supplied to the feedback function  $F_O()$ .

**Definition 9.4.6.** The output phase nonlinear feedback function  $F_O(st) : \mathbb{F}_2^{32} \rightarrow \mathbb{F}_2^8$  is defined as follows.

$$\begin{aligned}
 F_O(st) &= a + b + c + d \\
 \text{where} \\
 a &= \text{LookupTable}[s_a] \wedge 0xA0 \\
 b &= \text{LookupTable}[s_b \vee a_7] \wedge 0x09 \\
 c &= \text{LookupTable}[s_c \vee (b_0 \ll 2)] \wedge 0x12 \\
 d &= \text{LookupTable}[s_d \vee (c_1 \ll 5)] \wedge 0x44
 \end{aligned}$$

with  $s_a, s_b, s_c, s_d$  as given by Definition 9.4.2, and LookupTable as given by Appendix B.1.

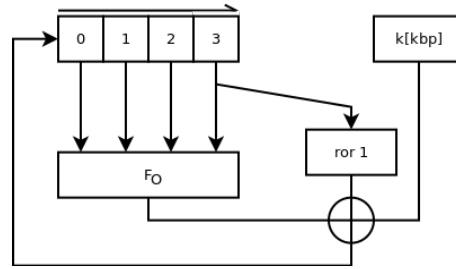


Figure 9.7: Phase 3: challenge absorption round function  $R_O(st, k, kbp)$ .

Round 3 function  $P3()$  resembles  $P2()$ , but the round function  $R_O()$  is invoked *twice* before the key byte pointer  $kbp$  is incremented. challenge related permutations. Second, in order to produce *two* bits of output, the round function is called 34 times, incrementing the key byte pointer  $kbp$  after each two rounds. The two output bits are defined as  $[st_{31}, st_{30}]$ . The round function is visualized in Figure 9.7.

The pseudocode for phase 3 is given in Code Listing 11.

---

**Code Listing 11** Phase 3

---

```

function P3( $st, k, kbp$ )
     $r \leftarrow 0$ 
    for  $i = 0$  to 15 do
        for  $j = 0$  to 16 do
             $st \leftarrow R_O(st, k, kbp)$ 
             $st \leftarrow R_O(st, k, kbp)$ 
             $kbp \leftarrow kbp + 1 \text{ mod } 29$ 
         $r_{47-(2*i)} \leftarrow st_{31}$ 
         $r_{46-(2*i)} \leftarrow st_{30}$ 
    return  $st, kbp, r$ 

```

---

Phase 3 computes the values for the response bits  $r_{47}$  down to  $r_{16}$ . The remaining bits are computed in phase 5.

#### 9.4.4 Phase 4: Remaining challenge absorption

The last 16 bits  $c_{15}$  to  $c_0$  of the challenge are absorbed in phase 4. The structure of this phase is identical to phase 2, using the same round function and nonlinear feedback function. The pseudocode is given in Code Listing 12.

This phase absorbs the challenge bits that have not been absorbed in Phase 2: bits  $c_{15}$  to  $c_0$ .

---

**Code Listing 12** Phase 4

---

```

function P4(st, c, k, kbp)
  for i = 32 to 47 do
    for j = 0 to 16 do
      st  $\leftarrow R_A(st, c_{47-i}, k, kbp)$ 
      kbp  $\leftarrow kbp + 1 \bmod 29$ 
  return st, kbp

```

---

### 9.4.5 Phase 5: Remainder of response generation

Phase 5 is constructed similarly to phase 3, generating the last 16 bits of the response. It employs the same round function structure and the same nonlinear feedback function. The pseudocode for phase 5 is given in Code Listing 13.

---

**Code Listing 13** Phase 5

---

```

function P5(st, k, kbp)
  r  $\leftarrow 0$ 
  for i = 16 to 23 do
    for j = 0 to 16 do
      st  $\leftarrow R_O(st, k, kbp)$ 
      st  $\leftarrow R_O(st, k, kbp)$ 
      kbp  $\leftarrow kbp + 1 \bmod 29$ 
    r47-(2*i)  $\leftarrow st_{31}$ 
    r46-(2*i)  $\leftarrow st_{30}$ 
  return st, kbp, r

```

---

This phase computes the response bits that have not been computed in Phase 3: bits  $r_{15}$  to  $r_0$ .

## 9.5 Properties of the cipher

### 9.5.1 Table lookups

The implementations we reverse-engineered use a lookup table (Appendix B.1) in the round function. The table is balanced for each output bit, that is, when comparing all entries in the table, the amount of 0 and 1 bits are equal for each bit position. The table has many duplicate entries and as such, is not an s-box. Rather, it implements a nonlinear function.

In the round function, four lookups are performed, and each time, two different bit positions of the retrieved value are used. We can thus replace the single 8-to-8 bit lookup table by four 8-to-2 bit lookup tables, which we will label according to the mask that is applied on the output value by the round function:  $T_{0xA0}$ ,  $T_{0x09}$ ,  $T_{0x12}$ ,  $T_{0x44}$ .

When inspecting the truth table of the first 8-to-2 bit table  $T_{0xA0}$ , we observe several relations. A Karnaugh map of the 8-to-2 bit function is found in Listing 9.2. The four topmost rows and four leftmost columns designate the values of the different bit positions of selector byte  $s_a$ , while the table shows the associated two bits<sup>11</sup>.

---

<sup>11</sup>In order to keep the map readable, only the two relevant bit positions are displayed, the most significant one first.

	3	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
	2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	1
	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	1
	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1
7 6 5 4																		
0 0 0 0 :	11	11	00	11	00	11	11	00	00	00	00	00	11	11	00	00		
0 0 0 1 :	00	00	11	11	00	11	00	11	11	11	11	11	00	11	00	11	00	
0 0 1 0 :	00	00	11	00	11	00	00	11	11	11	11	11	11	00	00	11	11	
0 0 1 1 :	11	11	00	00	11	00	11	00	00	00	00	00	11	00	11	00	11	
0 1 0 0 :	00	00	11	11	11	11	00	00	11	11	11	00	00	00	11	00		
0 1 0 1 :	11	00	00	00	11	00	11	00	00	11	00	11	11	11	00	11		
0 1 1 0 :	11	11	00	00	00	00	11	11	00	00	00	11	11	11	00	11		
0 1 1 1 :	00	11	11	11	00	11	00	11	11	00	11	00	00	00	11	00		
1 0 0 0 :	01	01	10	01	10	01	01	10	10	10	10	10	01	01	10	10		
1 0 0 1 :	10	10	01	01	10	01	10	01	01	01	01	01	10	01	10	01	10	
1 0 1 0 :	10	10	01	10	01	10	10	01	01	01	01	01	01	10	10	01	01	
1 0 1 1 :	01	01	10	10	01	10	01	10	10	10	10	10	01	10	01	10	01	
1 1 0 0 :	10	10	01	01	01	01	10	10	01	01	01	01	10	10	10	01	10	
1 1 0 1 :	01	10	10	10	01	10	01	10	10	01	10	01	01	01	10	01		
1 1 1 0 :	01	01	10	10	10	10	01	01	10	10	10	01	01	01	10	01		
1 1 1 1 :	10	01	01	01	10	01	10	01	01	10	01	10	10	10	01	10		

Listing 9.2: Karnaugh map of the Model C nonlinear feedback table.

A first observation is that bit 7 determines whether the second output bit matches or complements the first output bit<sup>12</sup>. Second, a flip in bit 5 always results in a flip of both output bits: we see each row where bit 5 is set is the complement of the the row where bit 5 is not set. These observations can be implemented as two simple linear operations, which allows us to eliminate bit 5 and 7 from the function input and conditionally perform the bit 5 and bit 7 linear operation a posteriori. This leads to a simplified table for  $T_{0xA0}$ , as shown in Listing 9.3.

	3	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
	2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	1
	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	1
	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1
7 6 5 4																		
* 0 * 0 :	1	1	0	1	0	1	1	0	0	0	0	0	1	1	0	0		
* 0 * 1 :	0	0	1	1	0	1	0	1	1	1	1	0	1	0	1	0		
* 1 * 0 :	0	0	1	1	1	1	0	0	1	1	1	0	0	0	1	0		
* 1 * 1 :	1	0	0	0	1	0	1	0	0	1	0	1	1	1	0	1		

Listing 9.3: Karnaugh map after fixing bit 5 and 7 to zero. The effect of these bits can be written as a linear function on the output.

The original value can be derived as follows. For selector  $s$ , look up bit  $b$  in the reduced table for index  $s \wedge 0b01011111$ . The two output bits can be obtained as  $[b \oplus s_5, b \oplus s_5 \oplus s_7]$ . The remaining three tables are similar: all have one bit that complements the output and a bit that defines the second output bit. We have thus reduced the nonlinear component from four large 8-to-8 bit lookups to four small 6-to-1 bit lookups.

### 9.5.2 Dependency between table lookups

In the feedback function  $F_A()$  and  $F_O()$ , all but the first table lookup are dependent on earlier lookup values. This is visualized in Figure 9.8a.

Similarly, the third lookup is dependent on the second, and the fourth is dependent on the third. However, when inspecting the structure of the mapping, it becomes apparent that this dependency can be rewritten as a linear operation on the output of the latter table lookup. This allows for simplification of the lookup function to a shape where the four lookups can be done independently, and the interdependency can be computed as a series of XOR operation a posteriori.

<sup>12</sup>Naturally, this is also valid the other way around, but the choice for which one to work with is arbitrary.

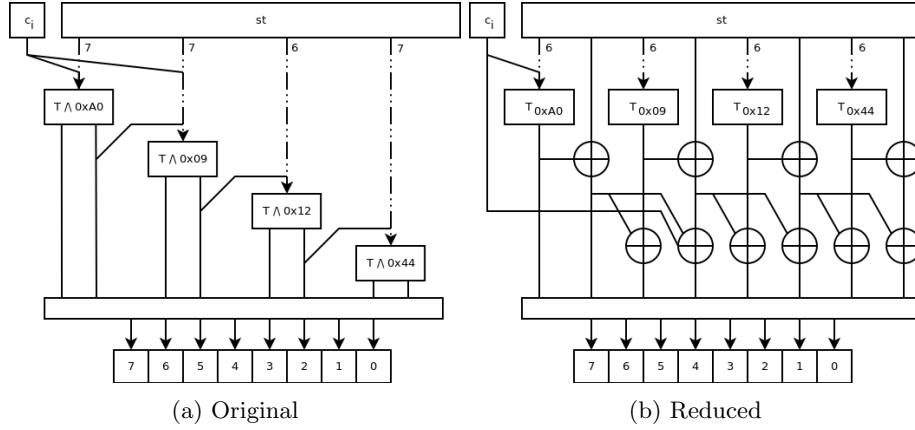


Figure 9.8: Removal of dependencies of nonlinear function inputs on earlier outputs. Mapping to output bits is not displayed. Dotted lines represent multiple bits, where the label indicates the number.

We can combine this with the structure we identified in the lookup tables in Section 9.5.1. The structure of the simplified  $F_A()$  function is visualized in Figure 9.8b,  $F_O()$  is of a

### 9.5.3 Inverse round function

Inverting the absorption round function will allow to retrieve the pre-round state  $st_{pre}$ , having knowledge of selected challenge bit  $c_i$ , post round state  $st_{post}$  and the currently selected key byte  $k[kbp]$ . Three of the pre-round state bytes are trivially recovered, as we simply have to invert the state shift operation. Recovery of the last pre-state byte  $st_{pre}[3]$  is harder, as it was shifted out of the state register. The known post-state byte  $st_{post}[0]$ , however, is partially derived from  $st_{pre}[3]$ , as was given in Definition 9.4.1. We can rewrite the computation of  $st_{post}[0]$  to obtain an equation for obtaining  $st_{pre}[3]$ :

$$\begin{aligned} st_{post}[0] &= ror(st_{pre}[3]) \oplus F_A(st_{pre}, c_i) \oplus k[kbp] \\ ror(st_{pre}[3]) &= st_{post}[0] \oplus F_A(st_{pre}, c_i) \oplus k[kbp] \\ st_{pre}[3] &= rol(st_{post}[0] \oplus F_A(st_{pre}, c_i) \oplus k[kbp]) \end{aligned}$$

As we assume the value of  $k[kbp]$  to be known, we only need to obtain the value of  $F_A(st_{pre}, c_i)$  to be able to compute  $st_{pre}[3]$ . As the nonlinear feedback function  $F_A()$  depends on the previous state, of which one byte is unknown, we have 256 candidate values. A candidate pre-state must be consistent with the post-state, as is denoted in the following lemma.

**Lemma 9.5.1.** *The set of pre-images  $S$  for the absorption phase round function with state  $st_{post}$  is given as follows.*

$$\begin{aligned} \forall st_{pre} \in \mathbb{F}_2^{32} : st \in S \text{ iff:} \\ st_{pre} &= (st_{post} \ll 8) + a \\ \text{for any } a \in \mathbb{F}_2^8 \text{ satisfying} \\ a &= rol(st_{post}[0] \oplus F_A(st_{pre}, c_i) \oplus k[kbp]) \end{aligned}$$

These pre-states  $st_{pre} \in S$  satisfy  $F_A(st_{pre}, c_i, k, kbp) = st_{post}$ . A very similar lemma can be formulated for the generation phase round function  $R_O()$ .

i	Round	$st_{30}$	$st_{32}$
0	34	15	8
1	68	31	24
2	102	8	9
3	136	24	25
4	170	9	10
5	204	25	26
6	238	10	11
7	272	26	27
8	306	11	12
9	340	27	28
10	374	12	13
11	408	28	29
12	442	13	14
13	476	29	30
14	510	14	15
15	544	30	31

Table 9.1: Bit origins per output moment

Generally, only a single  $a$  satisfies this equation, resulting in a single valid  $st_{pre}$ . However, state merges can occur, where  $F_A(st_a, c_i, k, kbp) = F_A(st_b, c_j, k, kbp)$  where  $st_a \neq st_b$  or  $c_i \neq c_j$ . In this case, both  $st_a$  and  $st_b$  will be a pre-image of  $F_A(st_a, c_i, k, kbp)$ .

#### 9.5.4 Rotate, round counts and array sizes

For this section, we need to define the notion of *origin*. Suppose we make a modification to the PCF7935 algorithm, leaving out the `xor` operation with  $k[kbp]$  and `xor` operation with feedback functions  $F_A()$  and  $F_O()$ . The algorithm then only performs bitwise and byte-wise rotates. If for any round  $i$  we say bit  $st_a$  originates from position  $p$ , we mean that  $st_p$  ended up at the position of bit  $st_a$  after  $n$  invocations of the round function.

It is interesting to know from where  $st_{31}$  and  $st_{30}$  originate when they are output as response bits. This is determined by the number of rounds between output moments (34), each round performing a right rotate by one byte, and an additional right rotate by one bit on the new  $st[0]$ . The table below shows for each output moment (where  $i$  relates to the phase 3 pseudocode in Section 9.4.3) which are the bit origins of the state bits  $st_{31}$  and  $st_{30}$ . The origin is defined as the bit position at the beginning of phase 3.

During challenge absorption, the round function  $R_A()$  is invoked 17 times per challenge bit while, during output generation, the round function  $R_O()$  is invoked 34 times. The choice for 17 and 34 instead of, for instance, 16 and 32, is motivated by the fact that if not, the state register would have shifted a number of times that is divisible by 4. This would imply that between each time output bits are generated, the state register would have made a number of full cycles. When 17 rounds are performed, the state register makes a *quarter* cycle more. This, combined with the rotate operation on  $st[3]$ , creates variation in which bit origins are output, as is visible in Table 9.1. However, a pattern is still visible, as each output moment, one bit position is re-used two output moments later.

As stated in Section 9.4.1, key byte array  $k$  has a repeating structure. While this may seem odd at first, choosing to increase the length to 29 bytes has a positive effect on the strength of the cipher. After each round (or 2 rounds during output phases) the key byte pointer  $kbp$  is incremented. The choice for a length of  $k$  that is relatively prime with 4, the number of state bytes, leads to a situation where each key byte is only xorred against a subset of state byte *origins*. This would introduce a weakness, potentially allowing for construction of a divide-and-conquer attack

by attempting to reconstruct the key in four separate parts<sup>13</sup>.

### 9.5.5 Two-phase challenge absorption / response generation

Due to the limited 32-bit internal state, absorbing all 48 challenge bits into the state at once would inevitably lead to a large number of collisions. As the challenge is 16 bits larger than the internal state, and no state divergence can occur during output generation, absorbing the full 48 bits would imply there are on average at least  $2^{16}$  challenges that lead to the same internal state, and consequentially, the same response.

Splitting the absorption / generation phases in a 32-bit and a 16-bit phase solves this<sup>14</sup>. However, there is a downside to this approach: the generated response can easily be split in two parts.

## 9.6 Attacks

Although it is certainly possible attacks against PCF7935 exist, we were unable to come up with an attack that would result in a reduced (less than  $2^{128}$ ) complexity. Further research would be required to shed light on the strength of the cipher. In order to justify the efforts spent on finding an approach to weaken the cipher, we will briefly motivate why we were unable to leverage several common attack classes<sup>15</sup> in order to carry out a successful attack against the PCF7935 cipher. Some remarks, that do not conceptually weaken the cipher but are potential security issues nonetheless, are discussed in Section 9.7.

### 9.6.1 Divide-and-conquer attack

A divide-and-conquer attack is not viable, as, plainly put, there is nothing to divide. Such an attack exploits the possibility to separate a hard problem into two (or more) subproblems, each of which have a lower complexity than the original problem. In the case of PCF7935, that would amount to identifying a way to attack parts of the key separately. This could be a subset of bytes in  $k$ , but it could also involve attacking certain bit positions within the set of key bytes. This is however not possible, as the full 128-bit key is used with each invocation of round function  $R_A()$  or  $R_O()$ . The rotate 1 operation on the old state byte guarantees that the bits within each state byte are also variable, making separation of the problem into less complex subproblems infeasible.

### 9.6.2 Correlation attack

A correlation attack focuses on exploiting statistically biased output in order to learn information about specific state bits. While the feedback functions  $F_A()$  and  $F_O()$  do exhibit bias if parts of the state are fixed<sup>16</sup>, this cannot be extended to an attack. This is because of the distance between output moments. If after round  $n$  we obtain the two most significant bytes of  $st[0]$ , we only obtain the next two output bits after round  $n + 34$ . That means 34 invocations of  $F_O()$  have occurred. Although we may derive some statistical information from the generated bits, the cryptographic relevance of this information is lost by the time the next output bits are generated.

<sup>13</sup>Naturally, there is still the feedback function that needs to be dealt with, but it would certainly weaken the cipher.

<sup>14</sup>Although, due to the nature of the algorithm, some collisions still occur.

<sup>15</sup>These are all cryptographic attacks listed and discussed in Section 3.1 of [41], with the exception of the malleability attack, which cannot be applied on a challenge/response mechanism.

<sup>16</sup>Known or assumed

### 9.6.3 Guess-and-determine attack

While the Model B algorithm allowed for the construction of a guess-and-determine attack, this is not the case for Model C. The main problem is that although we do get two bits of state information per output moment, guessing state information will not reveal any advantages. This is due to the bitwise **xor** operation that takes place:

$$\text{ror}(st[3]) \oplus F_O(st) \oplus k[kbp]$$

Guessing either parts of the state, or the key byte will not allow to distinguish between a correct and an incorrect guess. If we would guess the full state, we could predict the output of  $\text{ror}(st[3]) \oplus F_O(st)$ . We could compare this to two observed output bits  $st_{31}$ ,  $st_{30}$ . However, the **xor** step removes the distinguisher, as is shown below for :

$$st_{31} = \begin{cases} 0 & \text{if } s_7 \wedge k[kbp]_7 \\ 0 & \text{if } \overline{s_7} \wedge \overline{k[kbp]_7} \\ 1 & \text{if } s_7 \wedge \overline{k[kbp]_7} \\ 1 & \text{if } \overline{s_7} \wedge k[kbp]_7 \end{cases}$$

where  $s = \text{ror}(st[3]) \oplus F_O(st)$ . In other words, guessing  $s$  would<sup>17</sup> amount to finding the corresponding key bits  $k[kbp]_{7..6}$ , but the guess cannot be verified for correctness. This also holds when comparing multiple rounds, as  $kbp$  is incremented.

There is however, an exception, which will be detailed in the next section.

### 9.6.4 Differential cryptanalysis

During phase 2, the challenge is absorbed, and differential cryptanalysis would require some kind of exploitable relation in the state between related challenges to persist until the end of phase 2. This way, it could be identified by observing the output bits during phase 3. However, due to the large number of rounds, this relation is lost.

Possibly, the nature of state merges can be explored as a venue to weaken the cipher. State merges tend to occur during the last rounds of the absorption phase. As such, some information about the internal state can be derived: the conditions that result in a state merge were satisfied. Unfortunately, we have had insufficient time to fully explore this approach.

Another possible approach is an extension of the analysis made previously in Section 9.6.3. While in most cases, the **xor** operation with the key byte renders distinction of a proper guess of the state impossible, there is an exception to this. Five bytes of the key array  $k$  are known: at index 12, 13, 14, 15 and 28, the manufacturer-specific IDE bytes are found. The value of  $kbp$  is known throughout the algorithm. If we observe response bits generated at a time where  $k[kbp]$  is known, we do not only obtain two state bits, but also  $(\text{ror}(st[3]) \oplus F_O(st))_{[7..6]}$ . Considering the first 32 bits of response generation, the response bits  $r_{39}$ ,  $r_{38}$ ,  $r_{27}$ ,  $r_{26}$ ,  $r_{17}$  and  $r_{16}$  are generated in a round where a known key byte is selected. However, we have not been able to property this to a functional attack.

### 9.6.5 Algebraic attacks

As pointed out in 9.6.3, relations between observable output, internal state and key exist. These might be usable for an algebraic attack. However, there are several reasons why this is not feasible. Knowing or guessing the state would provide us with information about the associated key byte (at the time of producing output), however, any uncertainty about key or state bits would quickly propagate (well within 34 rounds) to no usable residual knowledge. Knowing or guessing parts

---

<sup>17</sup>Or guessing  $st$  and deriving  $s$

of the key would be a more fruitful approach, as the key is immutable and any knowledge will persist across rounds and across challenge/response pairs. However, the secret key is 128 bits large (although the key space of  $k$  is only 96 bits, see Section 9.4.1), and is too large to guess entirely. Guessing parts of the key leads to uncertainty about the state as rounds progress, thus the relation is insufficient to attack the cipher.

### 9.6.6 Meet-in-the-middle attacks

Although the algorithm can be run in the opposite direction, a meet-in-the-middle attack is not practical. Suppose we start at round  $n$ , and two bits of state information are revealed. Then, we consider the end state, where two more bits are revealed. In order to meet in the middle, one must be run forward for 17 rounds, while the other needs to run backwards for 17 rounds. One must then determine for which initial / post state pairs a valid path from start to end exists. This is, however, strongly dependent on the key array, and all *distinct* key bits are used within 17 rounds. Although a meet-in-the-middle *might* reduce the attack complexity compared with rolling forwards for 34 rounds, we did not find a way to reduce the complexity below (or even near)  $2^{128}$ . The fact that the algorithm doubles the amount of rounds between output moments, when compared to the challenge absorption phase, is a design choice that increased robustness against this class of attack.

## 9.7 Remarks

### 9.7.1 Usage in 32-bit mode

In Model C, only 32-bit challenges and 32-bit responses are used. This is done by computing only phases 1 to 3. This weakens the security of the authentication scheme, as it becomes less resilient against birthday attacks.

### 9.7.2 Random number generation

The Model C pseudorandom number generator is used in order to generate the challenge, and as such, should be both cryptographically secure and properly seeded. The design of the Model C PRNG uses a 32-bit multiplication constant that was originally used in the Netscape 1.1 PRNG, dating back to 1995, and proven insecure in 1996[20]. While the Netscape 1.1 PRNG uses time and process id as entropy source, the Model C ECM, visible in Figure 9.9, solely uses time for seeding the PRNG. Clearly, time is not a sufficient seed for a cryptographically secure pseudorandom number generator. The large multiplications with the Netscape 1.1 0xDEECE66D constant do not bear any cryptographic significance. Although a time-based side channel attack is outside the scope of this thesis, it is fair to assume this attack vector could result in breaking the randomness of the challenge, potentially allowing for a replay attack.

```

10 num_rounds = num_bytes;
11 random_bytes_used = 4;
12 if ( num_bytes )
13 {
14     dest_byte_ptr = dest - 1;
15     do
16     {
17         if ( random_bytes_used >= 4 )           // if random_bytes_used == 4, get 4 new random bytes
18         {
19             do
20             {
21                 asm
22                 {
23                     mftbu    r3                # Move from time base register (upper)
24                     mftb    r4                # Move from time base register (lower)
25                     mftbu    r5                # Move from time base register (upper)
26                 }
27             } while ( _R5 != _R3 );           // Check if consistent value was retrieved (no wraparound)
28
29             // XOR new random time value with old entropy pool,
30             // do large, overFlowing multiplications
31             LODWORD(v9) = _R4 ^ entropy_pool[1];
32             v10 = 0xDEECE66D * ( _R3 ^ entropy_pool[0] ) + 5 * v9;
33             v11 = 0xDEECE66D * v9 >> 32;
34             LODWORD(v9) = 0xDEECE66D * v9;
35             HIDWORD(v9) = v10 + v11;
36             v9 += 11164;
37             entropy_pool[0] = HIDWORD(v9);
38             HIDWORD(v9) = entropy_pool[0];
39             entropy_pool[1] = entropy_pool[0];
40             entropy_pool[0] = v9;
41             num_bytes = HIDWORD(v9) ^ v9;       // Note: num_bytes is now actually the new 32-bit random
42             random_bytes_used = 0;
43         }
44         --num_rounds;
45         ***dest_byte_ptr = num_bytes;
46         num_bytes >>= 8;
47         ++random_bytes_used;
48     } while ( num_rounds );
49 }
50 }
51 }

```

Figure 9.9: Model C ECM PRNG pseudocode. Lines 23-26 retrieve the value from the clock register. The entropy pool is not properly seeded after reset.

## Chapter 10

# Suggestions for improvement

Multiple approaches can be taken when designing an ECM authentication protocol. While Model A and Model C implement unilateral authentication, Model B implements mutual authentication. Unilateral authentication serves to allow the ECM to ascertain the BCM is authorized. In the encountered implementations, the BCM will only authenticate itself if the BCM recognizes the presence of an authorized key, so successful authentication of the BCM must also be considered a proof that an authorized key is present. A mutual authentication scheme would also authenticate the ECM towards the BCM. This has the advantage that simply replacing the original ECM would be detected, allowing the BCM to take measures to prevent the vehicle from operating. Car theft by replacing the ECM by a module with disabled immobilization functionality could thus be prevented.

Whether unilateral or mutual authentication is chosen, both the protocol and the underlying cryptographic primitives must be resilient to attack. A lot of academic work has been done in this field, and multiple light-weight authentication protocols have been published and scrutinized by the academic community. ISO 9798-2[17] specifies several variants of both unilateral and mutual authentication protocols, based on symmetric encryption and decryption, while ISO 9798-4[16] specifies protocols based on a cryptographic hash function. Lastly, ISO 9798-3[15] specifies authentication schemes using digital signatures, which, although generally more computationally intensive, has the advantage that secrets never have to be transmitted over the bus, even when a new module is installed in the vehicle.

Several academic publications have presented protocols for secure communication over CAN. LeiA[34], as presented by Radu et al. implements an authentication protocol for ECUs, allowing for light-weight authentication of CAN messages by means of message authentication codes. Kurachi et al. presented CaCAN[28], a centralized authentication system that relies on a single central monitoring node, that will destroy unauthorized CAN frames by transmitting an error frame during the transmission of the unauthorized frame<sup>1</sup>. More proposals can be found in the literature[39][21][23].

Besides using a secure protocol, a suitable choice for cryptographic primitives needs to be made. It is wise to choose a well known, publicly scrutinized cryptographic primitive such as SHA-256, SHA-3 and AES. Faster ciphers have been published, such as Chaskey, presented in [31], but while assessed by the scientific community, these have not yet withstood the amount of academic attention necessary to justify strong confidence in its security guarantees. Either way, the use of secret proprietary cryptographic components is to be avoided, as there are numerous examples of cryptographic systems that were broken shortly after their secrecy was lost[42][43][4].

It is important the cryptographic primitives result in a low latency on embedded platforms, in order to ensure swift authentication and immobilizer deactivation. Strong cryptographic primitives can often be efficiently implemented on embedded devices. For many architectures, optimized implementations of popular ciphers and hashing algorithms are publicly available[33][3].

---

<sup>1</sup>This is possible due to the distinction between the dominant and recessive CAN signals, as outlined in Section 3.3.

As an example of currently accepted latency, we will consider PCF7935. Its cryptographic algorithm invokes its round function 1632 times, and each round relies on a less-than-trivial feedback function. It is fair to assume suitable scrutinized cryptographic primitives exist that can meet or exceed PCF7935 in both latency and security.

# Chapter 11

## Discussion

In this section, we will elaborate how the work fits into the field of vehicle security and what contribution it seeks to make. Additionally, we will summarize the findings in order to formulate answers to the research questions specified in Section 1.

As was stated before, to the best of our knowledge, no academic research has explicitly investigated ECM authentication. By means of the three case studies, we hope to shed some light on the general shape and strength of this authentication step.

One might argue that ECM authentication is not crucial to the security of a vehicle. Naturally, all modern vehicles implement secondary security mechanisms, of either an electronic or mechanical nature. While mechanical door locks are becoming less popular, in favor of RKE (Remote Keyless Entry) systems, mechanical ignition locks are still commonly encountered in currently produced vehicles. A mechanical ignition lock generally also acts as a steering column lock. While the car can be powered in other ways than by turning the ignition to ‘on’<sup>1</sup>, theft of the vehicle requires - in one way or another - the steering column lock to be disengaged. Vehicles equipped with Keyless Go do no longer require the use of mechanical keys. The presence of a transponder in the vehicle is detected, and the column lock will be disengaged electronically. Still, these vehicles rely on more than ECM authentication for theft protection purposes. Nevertheless, both of these security mechanisms have been proven to provide less-than-perfect security. Mechanical locks can either be picked or broken, and copying mechanical keys is<sup>2</sup> relatively easy. Even a photograph of a mechanical key can be used to craft a duplicate. Numerous times, RKE and Keyless Go systems have been proven insecure, and it is not reasonable to depend only on these systems for theft prevention.

The investigated vehicles have been manufactured between 2008 and 2009, and as such, it is possible manufacturers have adopted more secure authentication mechanisms. However, we have found all three of the investigated protocols in currently produced vehicles. The development cycle of most car manufacturer seems to be relatively slow. Since ECM authentication has had little academic attention over the past few years, there is need for a public discussion regarding the strength of the involved protocols and cryptographic primitives. While manufacturers may be motivated to protect their vehicles against theft, it is possible insurance companies have a stronger incentive to secure the immobilizer system, as financial losses are suffered for each theft claim.

Also, it must be noted that new immobilization systems are not necessarily secure, or even more secure than its predecessor. Garcia et al. demonstrated this in [18], as they found a 2009 Remote Keyless Access system based on XTEA was using a single global key. All cars equipped with this system thus share the same secret key, nullifying the advantage of choosing strong cryptographic primitives.

---

<sup>1</sup>For instance, by “hotwiring” the wires connected to the ignition lock.

<sup>2</sup>Depending on the key and the resources of an attacker

## 11.1 How do manufacturers implement BCM-ECM authentication

Of the three investigated manufacturers, two use unilateral authentication, implemented as a challenge-response protocol employing a symmetric key. One implements symmetric key mutual authentication, where the BCM will only generate a valid response if the authenticity of the ECU is validated, ruling out the risk of a birthday attack. All encountered models use 32-bit challenges, the response length however varies. Two models employ 32-bit responses, while Model B was once more the exception with a 14-bit response length.

Another interesting note is that none of the three manufacturers use ISO-TP packets for inter-component authentication, but use raw CAN frame payloads instead. While not necessarily a problem, ISO-TP allows for the construction of multi-frame packets, which can in turn be used when handling longer challenge-response messages

## 11.2 What is the strength of the cryptographic components used in the BCM-ECM authentication

First, we have encountered two algorithms where the key space is insufficiently large. For Model A and Model B, the key space is severely lacking, rendering the algorithm vulnerable to brute-force attacks. Model C employs a 128-bits key, which is sufficiently strong to render exhaustive search infeasible.

Second, all three models use 32-bit challenges, which is insufficient. While PCF7935, found in Model C, supports 48-bits challenges, this feature was not used in Model C. The use of 32-bit challenges poses a potential security risk, as a birthday attack may successfully be mounted.

Model A uses 32-bit responses, but as the key space is only  $34^4$ , there are less than  $2^{23}$  possible responses<sup>3</sup>. Model C uses 32-bit responses<sup>4</sup>. Model B uses 14-bit responses. A larger response length would be advisable, as it incurs virtually no performance penalty and would result in a lower chance of successfully guessing a correct response.

The cryptographic primitives underlying the authentication protocol of model A and model B were broken. Car-only attacks allow to recover the vehicle security code and deactivate immobilization in approximately 7 seconds for model B and on average 15 minutes for model A. If an authorized key is present, recovery of the vehicle security code can be done for Model A in less than a second. Model C does not exhibit obvious flaws and was not broken. However, it is based on a discontinued transponder technology, introduced in 1994. Relying on such obsolete cryptographic components is questionable.

## 11.3 How can manufacturers improve upon the current strength of BCM-ECM authentication

There is sufficient academic work online that specify protocols and cryptographic primitives. Adopting publicly scrutinized protocols and cryptographic components could greatly enhance the strength of ECM authentication. Some obvious flaws, such as extremely small keys, should clearly be avoided: one can have a strong algorithm, but when using a key space of  $2^{32}$ , a practical attack is trivially found. Rate limiting may solve the potential problem of an exhaustive search. However, relying on such a mechanism introduces a potential security risk, as methods may exist that allow an attacker to bypass rate limiting<sup>5</sup>. As such, it is advisable to provide strong cryptographic resilience against attacks involving large numbers of authentication attempts.

---

<sup>3</sup>And in practice, a lot less, as was shown in the car-only attack against model A in Section 7.5.2

<sup>4</sup>Once again, PCF7935 supports 48-bit responses, but this was not used.

<sup>5</sup>Such as, attempting to reset the timeout by sending an UDS ECU reset command as soon as a response was rejected.

It is important to note that while the ECM-BCM authentication step can be improved by following above recommendations, changes in other parts of ECU firmware design are required in order to secure the system as a whole. Weak seed-key authentication algorithms would allow an attacker to open an authenticated diagnostic session and read memory contents or program keys, defeating the need to break the ECM-BCM authentication. Weak random number generators, depending only on time, will allow for timing-based side channel attacks, potentially allowing an attacker to have the ECM generate a challenge chosen by the attacker. It is reasonable to assume current ECUs are also vulnerable to other classes of side channel attacks, but most of these<sup>6</sup> need physical access to the ECU PCB. This reduces the practicality of such attacks in a real-world scenario, as the procedures involve more than merely capturing and injecting packets on the CAN bus.

---

<sup>6</sup>Such as power glitching, clock glitching and fault injection.

## Chapter 12

# Conclusions

Although the investigated models are no longer being produced, we have confirmed that all three uncovered protocols are still being used in some currently-produced vehicles. The three case studies are thus still representative for current vehicles, and as such, allow us to draw some conclusions about the methods manufacturers employ for BCM authentication, and the quality of the underlying cryptographic components. On one examined vehicle (Section 8), we found a car-only attack, deriving the vehicle security code within seconds. This allows for the authorization of a new key and subsequently, disable immobilization. On another vehicle (Section 7), we also found a car-only attack, which on average takes about 15 minutes to obtain the vehicle security code or disable immobilization directly. The immobilizer protocol of the third vehicle (Section 9) does not exhibit obvious flaws and was not broken, but emulates an obsolete security transponder originating from 1994. While no attack is presented in this paper, relying on obsolete proprietary cryptographic components is questionable[41]. As all three case studies revealed shortcomings in the immobilizer protocol, flaws can be expected to be present in other models from different manufacturers.

For car thieves, breaking the BCM-ECM authentication is a very convenient approach, as it is non-intrusive<sup>1</sup> and uses a standardized OBD-II connector and protocol. Furthermore, attacks can be carried out discreetly and with high reliability. As criminals have proven themselves capable of wielding high-tech devices in order to facilitate their theft of modern vehicles, weak ECM authentication does incur an increased risk of theft.

As a conclusion, the strength of BCM-ECM authentication protocols assessed during this research is of inconsistent and generally unsatisfactory quality. Usage of short keys and challenges, poorly designed algorithms and reliance on secret, proprietary cryptographic components pose a risk. We would recommend manufacturers to reconsider the design of their authentication protocols and embrace well-understood public cryptographic algorithms in order to ensure the expectations of customers regarding theft resilience are met.

---

<sup>1</sup>Besides the requirement of physical access to the car interior.

# Bibliography

- [1] ADAC. Wie sicher sind keyless-schliesysteme? — adac. <https://www.youtube.com/watch?v=xHCUpLBGIKQ>, 2016. 1
- [2] ADAC. Autos und motorrder mit keyless-schliesystem, die der adac illegal ffnen und wegfahren konnte”. [https://www.adac.de/\\_mmm/pdf/Keyless\\_Liste-gepr%C3%BCfte-Fahrzeuge%20mit%20Motorr%C3%A4dern%2020170518\\_257944.pdf](https://www.adac.de/_mmm/pdf/Keyless_Liste-gepr%C3%BCfte-Fahrzeuge%20mit%20Motorr%C3%A4dern%2020170518_257944.pdf), 2017. 1
- [3] Kubilay Atas, Luca Breveglieri, and Marco Macchetti. Efficient aes implementations for arm based platforms. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 841–845. ACM, 2004. 51
- [4] Alex Biryukov, Ilya Kizhvatov, and Bin Zhang. Cryptanalysis of the atmel cipher in secure-memory, cryptomemory and cryptorf. In *ACNS*, volume 11, pages 91–109. Springer, 2011. 51
- [5] B Cherowitzo. Linear feedback shift registers. <http://www-math.ucdenver.edu/~wcherowi/courses/m5410/m5410fsr.html>. 26
- [6] SP Diagnostics. Sp diagnostics website. <http://spdiagnostics.com/>. 18
- [7] EU Directive. 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 of the European Communities L*, 286:1–44, 1995. 1
- [8] 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. 5
- [9] St George Evans and Edward N Birkenbeuel. Automobile-theft preventer., April 8 1919. US Patent 1,300,150. 1
- [10] Horst Feistel. Cryptography and computer privacy. *Scientific american*, 228:15–23, 1973. 21
- [11] International Organization for Standardization. Iso 14229-1:2013 road vehicles – unified diagnostic services (uds) – part 1: Specification and requirements. <https://www.iso.org/standard/55283.html>. 7
- [12] International Organization for Standardization. Iso 14230-1:2012 road vehicles – diagnostic communication over k-line (dok-line) – part 1: Physical layer. <https://www.iso.org/standard/55591.html>. 6
- [13] International Organization for Standardization. Iso 15765-1:2011 road vehicles – diagnostic communication over controller area network (docan). <https://www.iso.org/standard/54498.html>. 6

- [14] International Organization for Standardization. Iso 9141:1989 road vehicles – diagnostic systems – requirements for interchange of digital information. <https://www.iso.org/standard/16737.html>. 6
- [15] International Organization for Standardization. Iso 9798-2 entity authentication mechanisms using digital signature techniques. <https://www.iso.org/standard/29062.html>, 1998. 51
- [16] International Organization for Standardization. Iso 9798-4 entity authentication mechanisms using a cryptographic check function. <https://www.iso.org/standard/31488.html>, 1999. 51
- [17] International Organization for Standardization. Iso 9798-2 entity authentication mechanisms using symmetric encipherment algorithms. <https://www.iso.org/standard/50522.html>, 2008. 51
- [18] 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 *USENIX Security Symposium*, 2016. 53
- [19] Thomas Giesler. The first international workshop on it-solutions for physical security: State of the art car access security systems. [http://www.codekey.com.ar/public/manuales/basic\\_transponder.pdf](http://www.codekey.com.ar/public/manuales/basic_transponder.pdf). 36
- [20] I Goldberg and D Wagner. Randomness and the netscape browser. *Dr. Dobbs Journal*, 1996. 49
- [21] Bogdan Groza, Pal-Stefan Murvay, Anthony Van Herrewege, and Ingrid Verbauwhede. Libracan: A lightweight broadcast authentication protocol for controller area networks. In *CANS*, pages 185–200. Springer, 2012. 51
- [22] The Guardian. Thieves target luxury range rovers with keyless locking systems. <https://www.theguardian.com/money/2014/oct/27/thieves-range-rover-keyless-locking>, 2014. 1
- [23] Oliver Hartkopp and R MaCAN SCHILLING. Message authenticated can. In *Escar Conference, Berlin, Germany*, 2012. 51
- [24] New Wave Instruments. Linear feedback shift registers - implementation, m-sequence properties, feedback tables. [http://www.newwaveinstruments.com/resources/articles/m\\_sequence\\_linear\\_feedback\\_shift\\_register\\_lfsr.htm](http://www.newwaveinstruments.com/resources/articles/m_sequence_linear_feedback_shift_register_lfsr.htm). 26
- [25] SAE International. Sae j1962 diagnostic connector equivalent to iso/dis 15031-3:december 14, 2001. [http://standards.sae.org/j1962\\_201207/](http://standards.sae.org/j1962_201207/). 5
- [26] Karl Koscher. *Securing Embedded Systems: Analyses of Modern Automotive Systems and Enabling Near-Real Time Dynamic Analysis*. PhD thesis, University of Washington, 2014. 9
- [27] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. Experimental security analysis of a modern automobile. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 447–462. IEEE, 2010. 9
- [28] R Kurachi, Y Matsubara, H Takada, N Adachi, Y Miyashita, and S Horihata. Cacan-centralized authentication system in can. In *Proc. escar 2014 Europe Conference, Hamburg, Germany*, 2014. 51
- [29] Abrites ltd. Abrites ltd. website. <http://abrites.com>. 9
- [30] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015, 2015. 9

- 
- [31] Nicky Mouha, Bart Mennink, Anthony Van Herrewege, Dai Watanabe, Bart Preneel, and Ingrid Verbauwhede. Chaskey: an efficient mac algorithm for 32-bit microcontrollers. In *International Workshop on Selected Areas in Cryptography*, pages 306–323. Springer, 2014. 51
- [32] Karsten Nohl. Car immobilizer hacking. Sigint13, 2013. 9
- [33] Dag Arne Osvik. Fast embedded software hashing. *IACR Cryptology ePrint Archive*, 2012:156, 2012. 51
- [34] Andreea-Ina Radu and Flavio D Garcia. Leia: A lightweight authentication protocol for can. In *ESORICS (2)*, pages 283–300, 2016. 51
- [35] Florian Sagstetter, Martin Lukasiewicz, Sebastian Steinhorst, Marko Wolf, Alexandre Bouard, William R Harris, Somesh Jha, Thomas Peyrin, Axel Poschmann, and Samarjit Chakraborty. Security challenges in automotive hardware/software architecture design. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 458–463. EDA Consortium, 2013. 9
- [36] BJ Shinde, SS Kore, and SS Thipse. Comparative study of on board diagnostics systems - eobd, obd-i, obd-ii, iobd-i and iobd-ii. *International Research Journal of Engineering and Technology*, 3, 2016. 5
- [37] Simon Touch Automotive Solutions. Jlr can adapter to unlock the car and program via can-bus. <https://www.keyprogtools.com/key-programming/range-jlr-can-adapter>. 10
- [38] Auto Express UK. Keyless car crime up as gangs target vans. <http://www.autoexpress.co.uk/car-news/consumer-news/90328/keyless-car-crime-up-as-gangs-target-vans>, 2015. 1
- [39] 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. 51
- [40] Jan C van Ours and Ben Vollaard. The engine immobiliser: A non-starter for car thieves. *The Economic Journal*, 2016. 1
- [41] Roel Verdult. *The (in) security of proprietary cryptography*. SI: sn, 2015. 47, 56
- [42] Roel Verdult, Flavio D Garcia, and Josep Balasch. Gone in 360 seconds: Hijacking with hitag2. In *Proceedings of the 21st USENIX conference on Security symposium*, pages 37–37. USENIX Association, 2012. 9, 51
- [43] Roel Verdult, Flavio D Garcia, and Baris Ege. Dismantling megamos crypto: Wirelessly lockpicking a vehicle immobilizer. In *USENIX Security*, pages 703–718, 2013. 9, 51

# Appendix A

## Model B lookup tables

### A.1 FeedbackTable

Index								
0	66	66	66	66	99	99	99	99
8	66	66	66	66	99	99	99	99
16	99	99	99	99	66	66	66	66
24	99	99	99	99	66	66	66	66

### A.2 OutputTable

Index								
0	D7	74	24	91	83	65	BC	4D
8	E3	55	38	FB	76	2C	8D	70
16	F5	23	85	E5	8C	1C	FC	6E
24	A7	22	B9	33	39	7C	48	0A

## Appendix B

# Model C PCF7935 lookup table

### B.1 LookupTable

Index																
0	A8	A1	45	BA	5E	F7	BA	01	4E	47	03	18	F5	B8	11	4E
16	13	5E	BA	E5	45	A8	4C	F7	F5	B8	FC	03	AA	47	E7	18
32	5F	00	F6	1B	ED	12	09	E4	B9	E6	F4	B9	02	19	A2	EF
48	A0	BB	4D	44	B2	4D	BB	56	46	5D	0B	A2	5D	A2	54	FD
64	45	4C	BA	E5	B3	BA	45	5E	AA	A3	F5	4E	18	55	EE	11
80	EC	13	57	08	BA	45	A1	1A	0A	F5	11	EE	F5	AA	0A	F5
96	A0	FF	09	56	12	4D	F6	A9	4F	10	02	FD	FD	E6	5D	A2
112	5F	E4	B2	BB	4D	B2	44	A9	B9	02	F4	5D	02	5D	AB	02
128	73	7A	8C	61	8C	25	73	DA	95	9C	CA	C3	27	6A	D8	95
144	C8	85	61	3E	9E	73	97	2C	2E	63	27	D8	71	9C	3C	C3
160	96	DB	2D	D2	2D	C0	D2	2D	70	3D	2F	70	C2	CB	79	26
176	69	72	96	8D	7B	84	60	9F	8F	94	D0	6B	94	6B	8F	34
192	8C	97	73	2C	73	68	8C	97	63	78	3C	87	D8	87	27	D8
208	3E	C1	9E	C1	61	9E	61	DA	D1	2E	D1	2E	2E	71	CA	35
224	7B	24	C0	9F	C0	9F	3F	60	94	CB	CB	34	2F	34	94	6B
240	9F	36	69	60	84	69	96	7B	70	D9	26	8F	CB	86	79	D0