Beneath the Bonnet: A Breakdown of Diagnostic Security

Jan Van den Herrewegen(✉) and Flavio D. Garcia

University of Birmingham, Birmingham, UK
{jxv572,f.garcia}@cs.bham.ac.uk

Abstract. An Electronic Control Unit (ECU) is an automotive computer essential to the operation of a modern car. Diagnostic protocols running on these ECUs are often too powerful, giving an adversary full access to the ECU if they can bypass the diagnostic authentication mechanism. Firstly, we present three ciphers used in the diagnostic access control, which we reverse engineered from the ECU firmware of four major automotive manufacturers. Next, we identify practical security vulnerabilities in all three ciphers, which use proprietary cryptographic primitives and a small internal state. Subsequently, we propose a generic method to remotely execute code on an ECU over CAN exclusively through diagnostic functions, which we have tested on units of three major automotive manufacturers. Once authenticated, an adversary with access to the CAN network can download binary code to the RAM of the microcontroller and execute it, giving them full access to the ECU and its peripherals, including the ability to read/write firmware at will. Finally, we conclude with recommendations to improve the diagnostic security of ECUs.

1 Introduction

The functionality of a modern road vehicle is determined by a few dozen Electronic Control Units (ECUs). These ECUs are interconnected via one or several Controller Area Network (CAN) [10] buses. Powerful diagnostic protocols are put in place by the manufacturer to update or patch the vehicle in case of malfunction. The most prevalent diagnostic standards are Unified Diagnostic Services (UDS) [11] and its predecessor, Keyword Protocol 2000 (KWP2000) [12], which provide manufacturers and service technicians with advanced diagnostic features such as upload and download functionality. The main diagnostic access control mechanism is the so called 'seed-key protocol', a challenge-response protocol used to authenticate diagnostic devices. Even more sophisticated diagnostic protocols such as the Universal Measurement and Calibration Protocol (XCP) [1] enable service technicians to fully fine-tune ECUs. The functionality provided by XCP goes beyond that of traditional diagnostic protocols found in ECUs, which a knowledgeable attacker could abuse to take control of an ECU over CAN.

This work was partly funded by EPSRC Fellowship EP/R008000/1.

In many cars, diagnostic communication occurs on the CAN bus available on the OBD-II [13] port, which every vehicle commissioned in the European Union since 2004 [5] must be equipped with. However, the automotive network was never designed with an adversary in mind: the CAN bus is an unencrypted and unauthenticated network. Thus, ECUs cannot distinguish diagnostic messages originating from a diagnostic client from messages sent by an adversary. Previous research has indicated that individual ECUs connected to the internal network of a modern car can be compromised [17,18]. This becomes even more worrying when combined with a remote exploit, as demonstrated in [24]. After having gained access to the Telematics Unit, Valasek and Miller managed to remotely control crucial functionality of the car. With advanced features such as in-vehicle connectivity becoming the norm in modern cars, the automotive industry needs to shift towards better diagnostic security in ECUs.

Our Contribution. The contribution of this paper is three-fold:

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

Related Work. With its transformation towards more complex vehicular systems, the automotive industry is no stranger to cryptographic attacks against several security mechanisms it has put in place. Bogdanov first attacked the KeeLoq block cipher, which is used in various automotive anti-theft mechanisms, in [2], while further attacks on this cipher appear in [2,9,14]. Furthermore, an immobiliser system is in place to prevent an attacker from starting the car without a valid key fob. Ciphers used in the immobiliser system in various cars include the DST40 cipher, Megamos Crypto and the Hitag2 cipher, attacked respectively by Bono et al. in [3], Verdult et al. in [26,28] and in [27].

Several papers in the literature assess diagnostic security in ECUs, beginning with the work of Koscher et al. [17], which experimentally tests the security and capabilities of ECUs. The authors tamper with several safety-critical ECUs by sending diagnostic messages, while they reprogram the telematics unit to act as a bridge between the high-speed and low-speed CAN bus in the vehicle. The reprogramming of the unit consists of downloading code to its RAM memory and executing it. No access control mechanism was implemented on the studied ECU's.

Additionally, Miller and Valasek reverse engineered the complete reprogramming procedure on two Ford ECUs by analysing a diagnostic tool [18]. The tool reprograms the units by downloading a piece of code to the RAM memory of the microcontroller, which subsequently handles the reflashing of the unit. The authors abuse this mechanism to execute their own code on the ECU. After authenticating to the ECU, the authors use several diagnostic primitives to download the code to the unit. Once certain prerequisites have been met, a different diagnostic service makes the ECU jump to the downloaded code and thus execute it. No access restrictions are in place on the microcontroller, giving the code full access to peripherals such as the CAN bus.

Finally, Khan [15] raises several issues on security in the UDS protocol, the access control mechanism in particular. The paper states multiple security flaws in the security access service provided by UDS, more specifically on challenge generation and the complexity of the employed cipher. Furthermore, Khan notes that an attacker can recover the cipher and secret keys from the firmware.

Overview. The rest of this paper is organised as follows. Section 2 gives an overview of the most prevalent diagnostic protocols we have encountered in ECUs. In Sect. 3 we follow up with a description of the ciphers used in the diagnostic access control mechanism in several ECUs and propose ways to bypass these. In Sect. 4 we propose a method to remotely execute code on an ECU when having access to the CAN bus, of which we demonstrate the capabilities in Sect. 5. We discuss our findings in Sect. 7, while we suggest countermeasures and mitigations in Sect. 6. Finally, we conclude in Sect. 8.

2 Background

This section summarizes the most prevalent diagnostic protocol in ECUs, namely the Unified Diagnostic Services. We have encountered its predecessor, the Keyword Protocol 2000 in older ECUs, but since both protocols are very similar, we will only outline the main features of UDS. Note that we use the concepts *tester* and *client* interchangeably, both denoting the diagnostic device querying the ECU. Moreover, we briefly introduce diagnostic communication channels and summarize the main features of the XCP protocol.

2.1 Unified Diagnostic Services

The UDS standard defines several diagnostic sessions: in the default session an ECU executes its normal function in the internal vehicular network. A diagnostic client can change the active session with the *DiagnosticSessionControl* service to either a programming session or an extended diagnostic session, however the available functionality in these sessions is left up to the manufacturer's discretion.

The main access control mechanism is a challenge-response (also known as *seed-key* in automotive terminology) protocol specified by the *SecurityAccess* service, as depicted in Fig. 1. In order to authenticate to the ECU, a diagnostic client must send a *challenge request* to the ECU, which subsequently replies with a randomly generated challenge (also called the *seed* in automotive terminology).

Both the client and ECU calculate a *response* (also called the *key* in automotive terminology) from this challenge according to a manufacturer-specific cipher, based on a shared secret. The client is authenticated if it supplies the ECU with a valid response. Multiple security levels are defined in UDS, which the manufacturer is free to use for different levels of access. UDS only specifies the challenge-response protocol, leaving the choice of the cipher up to the manufacturer.

A tester can use the *RoutineControl* service to execute preprogrammed functions in the ECU, with each routine uniquely defined by a two byte identifier. The client can pass arguments in a routine control call if needed. The standard specifies some routines and their respective identifiers, such as the *EraseMemory* routine with identifier `0xFF00`, while the identifier range `0x200-0xC000` is reserved for manufacturer specific use.

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

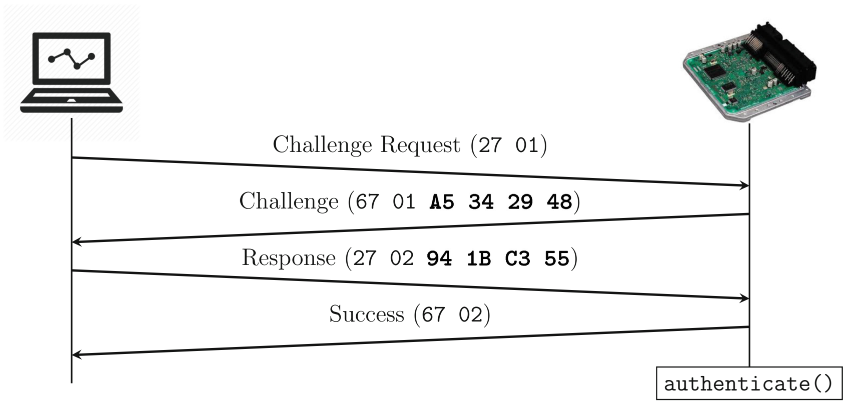
Challenge Request (27 01)
Challenge (67 01 A5 34 29 48)
Response (27 02 94 1B C3 55)
Success (67 02)
authenticate()

Fig. 1. The challenge-response protocol specified by UDS

2.2 Diagnostic Communication Channels

Neither UDS nor KWP 2000 specify the exact nature of the diagnostic communication channel (namely on which CAN ID each ECU listens for diagnostic messages). This is manufacturer specific, although there are some similarities across manufacturers. Since CAN frames with a lower identifier have priority over those with a higher identifier on the bus, diagnostic CAN identifiers are usually within the range `0x700-0x7FF`. Additionally, there is generally a clear relation between the CAN ID on which an ECU receives diagnostic messages, and on which ID it replies (e.g. $\text{ID}_{send} = \text{ID}_{recv} + 8$).

2.3 Universal Measurement and Calibration Protocol

Both XCP and its predecessor, the CAN Calibration Protocol (CCP) [16] are standardized by the Association for Standardization of Automation and Measuring Systems (ASAM). XCP is an application protocol which defines advanced features such as arbitrary read/write access to variables in ECU memory, synchronous data acquisition and flash programming of ECUs for development purposes. A diagnostic tool, also called the *master* in XCP, can analyse the connected ECUs, or *slaves*, through various XCP commands specified in the standard. The master has access to variables in memory by way of an ECU description file exclusive to each ECU, and can even download a reflashing kernel to the RAM for reprogramming purposes. Automotive software companies such as Vector Informatik support XCP solutions for ECUs of over 30 major automotive manufacturers [25], alluding to the extensive use of the XCP standard in the automotive industry.

3 Cryptanalysis of Diagnostic Protocols

In this section we analyse the ciphers used in the diagnostic challenge-response protocol, which we extracted from ECUs of three different automotive manufacturers. We recovered and analysed the firmware of 13 ECUs in total, comprising 8 different car models. We focused our efforts on ECUs with a security critical function, such as the Instrument Cluster and Body Control Module (which handle immobiliser functionality and store its secret keys), a Gateway (which separates the critical high speed CAN bus from other low speed buses), and a Telematics Unit (which provides connectivity to the outside world). Next, we revisit the cipher first described by Valasek and Miller in [18] and present new vulnerabilities, making it easy to circumvent in practise. Using the IDA Pro disassembler we have recovered challenge-response ciphers from the firmware of Ford, Volvo, Fiat and Audi ECUs. We present these ciphers and analyse their security.

3.1 Obtaining and Analysing ECU Firmware Images

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

Notation and Variables. To avoid any ambiguity, we will use the following notation in this section. C denotes the random challenge generated by the ECU, whereas R denotes the corresponding response. v_i denotes bit i of a variable v, with v_0 being the least significant (rightmost) bit, whereas $v[i]$ denotes byte i of v, with $v[0]$ being the most significant (leftmost) byte. $v \lll i$ refers to a rotation of v by i bits to the left. Finally, (v, w) denotes a concatenation of bytes v and w, with w the least significant byte.

3.2 Analysis of the Ford Challenge-Response Cipher

In this section we perform a cryptanalysis of the Ford cipher, which we have located in the firmware of several Ford ECUs but also in some Volvo units through our reverse engineering efforts. We introduce the cipher and demonstrate how an attacker can break it by means of an attack over CAN. We have found this cipher in the ECUs shown in Table 1.

Table 1. ECUs on which we examined and identified the Ford cipher

Make	Year	Model	ECU
Ford	2010	Focus MK2	Body control module Instrument Cluster
	2012, 2014, 2016	Focus MK3	Body control module Instrument cluster
	2008	Fiesta MK6	Instrument cluster
	2013, 2014, 2015, 2017	Fiesta MK7	Instrument cluster Body control module
Volvo	2015	V50	Telematics unit

Cipher Details. Both the challenge and response are three bytes in Ford ECUs. The cipher uses a slightly modified version of the Galois Linear-Feedback Shift Register (Galois LFSR) with an internal state of 24 bits, which is initialised with a constant (`0xC541A9`) stored in the firmware of the ECU. The output bit of the LFSR is XORed with a bit from a 64-bit input register R consisting of a 40-bit *secret* S and the 24-bit challenge C. Figure 2 depicts the structure of the modified Galois LFSR, while Definition 1 details the input bit of the cipher in round i. The cipher runs for 64 rounds: in the first 24 rounds, the challenge is shifted into the internal state, after which the cipher absorbs the 40-bit secret into its internal state. In each round, the XOR of the output bit of the LFSR and the input bit of the register is fed back into the tapped bits. The final response is derived from the 24-bit LFSR-state by permuting the nibbles of the state, as shown in Definition 2.

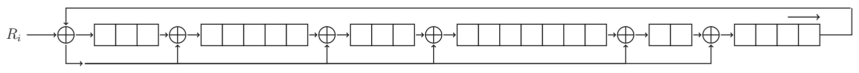

Fig. 2. Structure of the Ford LFSR

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

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

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

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

Weaknesses. The internal state of the Galois-LFSR used in the Ford algorithm contains merely 24 bits of entropy. What is even worse, we have observed the same start state and tapping sequence across *all* ECUs we have studied. With no added entropy from a varying start state or tapping sequence, only the 40-bit secret is unknown to an attacker. Through empirical tests we discovered that only the first 24 secret bits shifted into the internal state add entropy. In the subsequent 16 rounds we can set the input bit to zero, making the cipher a standard Galois-LFSR. One valid challenge-response pair enables an attacker to retrieve 24 bits of the secret, and thus recover the structure of the cipher. The attacker can obtain a valid challenge-response pair by making a diagnostic device authenticate to the ECU, which Valasek and Miller demonstrated in [18]. The cipher, however, can be broken even without knowledge of a challenge-response pair.

Attack over CAN. We demonstrate how an attacker can recover the secret used in the Ford cipher for a particular ECU without knowledge of any successful authentication pairs. Access to the diagnostic interface of the ECU is the only prerequisite for this attack.

Delay Mechanism. Unified Diagnostic Services specify an error code which indicates a delay timer is active on the ECU in case of too many failed security access attempts. The specifics of this mechanism are left up to the manufacturer. Many

ECUs implement this delay functionality and disable the security access service temporarily after a certain amount of failed attempts. An attacker can bypass this by requesting a soft reset using the *ECUReset* diagnostic service, which resets all timers and variables. Following a reset the attacker must request a new diagnostic session before they can request a new challenge.

Recovering Diagnostic Secrets on Ford and Volvo ECUs. We conducted our attack both on a 2012 Ford Body Control Module (BCM) and a 2015 Volvo Telematics Unit. These particular units do not implement the delay mechanism after a failed security access attempt. Once we request a diagnostic programming session, the units remains in programming mode until no further diagnostic messages are detected for a certain period ($\sim$5 s). Each security access attempt requires four CAN messages: a challenge request and reply followed by a response and a final message indicating whether the response was valid or not. All CAN frames are 8 bytes for the Ford diagnostic packages, making a physical CAN frame on the bus 135 bits in the worst case, with stuffing bits taken into account [20]. On the BCM, the diagnostic interface is available on the high speed CAN network, which runs at 500 kbit/s. One security access attempt takes four CAN frames or maximum 540 bits, so with a bitrate of 500 kbit/s that makes for a minimum of 1.08 ms per attempt, calculation time or other delays not taken into account. Since we reduced the complexity from 2^{40} of a brute-force attack to only 2^{24} attempts, this results in a search time of approximately 5 h in the best case scenario. Due to all other delays, the attack we implemented took approximately 15 h. We would like to emphasise that, since all ECUs use the same secret, an attacker only needs to do this once.

3.3 Analysis of the Fiat Challenge-Response Cipher

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

Cipher Details. Both the challenge and response are 32-bit in the Fiat implementation of the security access service. The cipher uses two 16-bit LFSRs, both with the structure depicted in Fig. 3. Both LFSRs absorb one input bit in each round, as detailed in Definition 4. The cipher runs for 24 rounds: in the first 8 rounds different constants ($S[0]$ and $S[2]$) are shifted into each state, whereas in the remaining 16 rounds the cipher absorbs one bit of the preprocessed challenge bytes into the state. Finally, the 32-bit response is derived from the LFSRs by combining the 16-bit internal states.

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

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

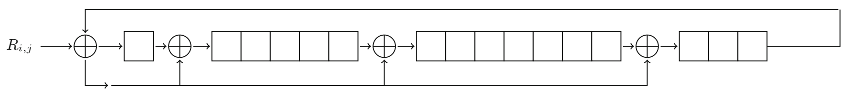

Fig. 3. Structure of the Fiat LFSR

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

$$R_{i,0} = (C[0] \oplus S[1], C[2] \ggg 5, S[0])_i$$
$$R_{i,1} = (C[3] \oplus S[3], P_2(C[1]), S[2])_i$$

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

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

Table 2. Secrets found in the firmware of two different Fiat ECUs

ECU	$S[0]$	$S[1]$	$S[2]$	$S[3]$	Taps	Start state
Fiat BSI 2012+	`0x12`	`0xDC`	`0x34`	`0x7A`	`0x8408`	`0xFFFF`
Fiat BSI 2012−	`0x21`	`0xDC`	`0x43`	`0x7A`	`0x3423`	`0xFFFF`

3.4 Analysis of the Volkswagen Group Cipher

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

Code listing 1. Audi gateway challenge-response algorithm

```
function CHALLENGE-RESPONSE(C)                        ▷ With C - 32-bit challenge
    $S = C$
    for $i$ in $\{0 \ldots 10\}$ do
        $S = S \lll 1$
        $feedback = S \wedge 1$
        if $i \in \{0, 2, 6, 7\}$ then
            if $feedback == 1$ then                    ▷ For rounds 0, 2, 6 and 7
                $S = S \wedge (\sim 1)$                 ▷ Clear the feedback bit
                $S = S \oplus$ 0x04C11DB7               ▷ XOR the tapped bits
            end if
        else
            if $feedback == 1$ then
                $S = S \oplus$ 0x04C11DB7
            else
                $S = S \mid 1$                          ▷ Set the feedback bit
            end if
        end if
    end for
    return $S$
end function
```

Cipher Details. Code Listing 1 details the cipher, which runs for 10 rounds. In each round, the cipher rotates the state to the left. The cipher is a standard Galois LFSR: if the feedback bit is set, a constant (the tapped bits, i.e. `0x04C11DB7` in the code below) is XORed into the state. Depending on the round, the feedback bit is either set or cleared.

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

Additionally, we identified a supplementary security issue in the firmware of this particular unit: if the diagnostic client provides an invalid response, the ECU performs an extra check, which compares the response to a hardcoded value (i.e. `0xCAFFE012`). The diagnostic tool is authenticated if it provides this value as the response. Regardless of existing vulnerabilities in the cipher, a hardcoded backdoor on the ECU introduces extra security implications.

4 Remote Code Execution over CAN

The ciphers we studied in Sect. 3 are in place to protect the ECU from unauthorised access. Once a diagnostic device is authenticated, the ECU unlocks priviliged diagnostic functionality, part of which allows executing more advanced diagnostic protocols like XCP. Despite its widespread use in the automotive industry, we failed to locate the XCP protocol in the firmware of the ECUs we studied. Instead, we found that the Original Equipment Manufacturer (OEM) enables a download of the XCP stack to the RAM of an ECU through various diagnostic services. Piggybacking on this required functionality for the XCP protocol, we have identified a generic approach to execute arbitrary code on an ECU over the CAN bus. Through our own reverse engineering efforts we have encountered this mechanism in ECUs made by several manufacturers. Provided that an attacker can bypass the access control mechanism of the diagnostic protocol as we showed in Sect. 3, the only prerequisite is that they can send and receive messages on the CAN bus. An attacker with access to the OBD-II port or who has compromised an ECU on the network, such as the Telematics Unit, can abuse this functionality to control or reprogram additional ECUs.

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

Downloading Sequence. Figure 4 shows the sequence of diagnostic messages required to execute the secondary bootloader on an ECU. Firstly, the diagnostic client must request a programming session. Until the client authenticates itself to the ECU, any necessary functionality remains unavailable. Once authenticated, the client can carry out certain checks and assertions about the ECU. These usually include reading out the software version and part number of the module as the secondary bootloader is dependent on the microcontroller. The client can transfer the secondary bootloader to the ECU through the download services provided by the running diagnostic protocol. Finally, the client requests a routine control either before or after the download (dependent on the manufacturer) in order to redirect the program flow to the secondary bootloader, which now resides in RAM.

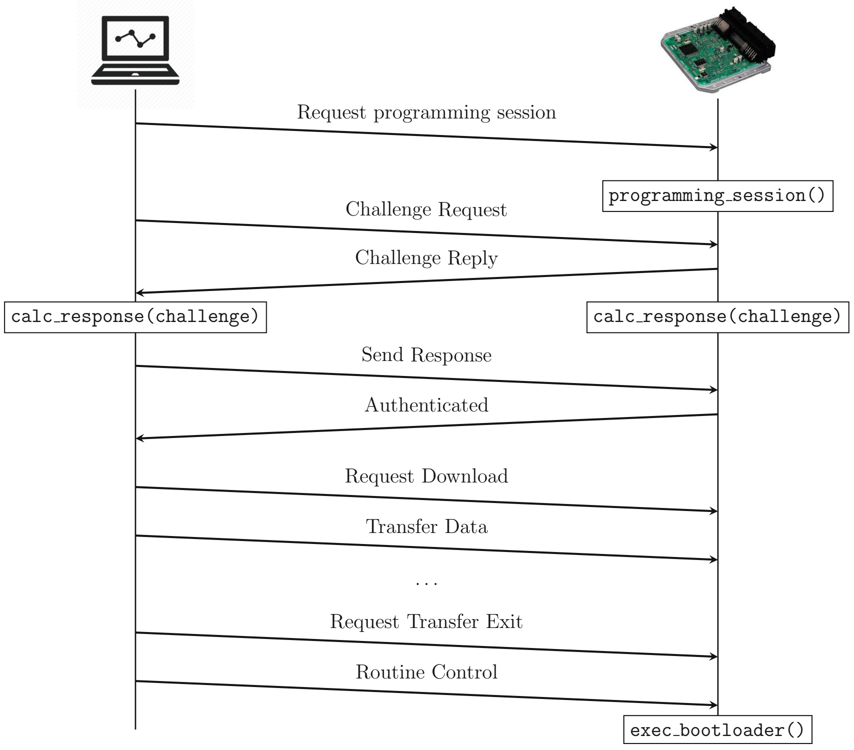

Fig. 4. Execution of the secondary bootloader

Memory Limitations. The ECU only provides a small area in RAM for the secondary bootloader, which usually suffices if the downloaded code performs a simple task (such as updating a variable in memory). Otherwise, the bootloader can download additional code into the RAM of the unit over the CAN bus.

4.1 Use Case: Changing the Odometer on a Ford Instrument Cluster

We have managed to change the odometer value on a 2016 Ford Focus Instrument Cluster (IC) through the secondary bootloader. The download of the secondary bootloader goes as follows for all Ford and Volvo ECUs we have analysed. With the ECU in a programming session and our device authenticated, we send a *requestDownload* message. The request has two arguments: the download address, which is located in RAM, and the size of the bootloader. If the microcontroller uses a paging mechanism, the address consists either of a page number and address within the page, or a physical address. Subsequently, we can transfer the secondary bootloader using the *transferData* service, after which the ECU expects a *requestTransferExit* message. Finally, to execute the downloaded code, we must send a *routineControl* message. Arguments to this message are the routine identifier, which is `0x301`, and the exact address where the microcontroller should jump to. The mileage on this Instrument Cluster is stored on an external Electrically Erasable Programmable Read-Only Memory (EEPROM) chip, namely the M95320 manufactured by ST Microelectronics. The main microcontroller, a Renesas μPD70F3425, is connected to the EEPROM chip through a Serial Peripheral Interface (SPI). Once we identified the pins used for the serial communication with the EEPROM chip, we managed to arbitrarily reduce the mileage by writing the desired value to the memory locations where the mileage is stored. Multiple ECUs store the mileage in a modern car, meaning that an attacker must repeat this process for all relevant ECUs if he wishes to successfully tamper with the mileage in a car.

It should be noted that Valasek and Miller first documented this bootloader mechanism to reprogram a Ford Smart Junction Box in [18]. There are several differences to the sequence denoted above compared to what Valasek and Miller describe. Firstly, the address the authors specify in the download request to the ECU is zero, which makes the ECU download the code to a predefined address in RAM. Subsequently, the authors call a routine control with identifier `0x304`, making the ECU jump to the same predefined address as the download. Finally, the code is only executed if the first four bytes of the secondary bootloader are equal to a value stored in the firmware of the ECU. We have only encountered this 'security' feature in one of the Ford ECUs we analysed.

4.2 Use Case: Reprogramming a Fiat Body System Interface

We have analysed the reprogramming process for both a current Delphi Fiat Body System Interface (BSI) and its predecessor, which are deployed in a range of Fiat vehicles. Execution of the secondary bootloader goes as follows for both Fiat BSIs. The ECU must be in a programming session and 'unlocked' for security level 1, following the steps from Sect. 3. In order to execute the downloaded code after the download, we must first write the identifiers with ID's `0xF184` and `0xF185` through the *writeDataByIdentifier* service. This sets a flag in memory necessary for the following routine control to complete successfully. Next, we must execute the *eraseMemory* routine control with arguments the identifier (`0xFF00`), the start address and end address of the memory area in RAM to which we will download the code. In order to make the microcontroller jump to the code, it is crucial that this range is equal to the size of the downloaded data. Otherwise, the download will terminate normally but will not result in a jump to RAM. If all prerequisites described above are met, the microcontroller will jump to a predefined address in RAM after the last *TransferData* request.

This address is set in the firmware and is dependent on the memory layout of the microcontroller as the bootloader always resides in RAM. Hence, in order to redirect the program flow to our code, this predefined address must be contained within the download range of the bootloader. Trace 1 shows the required diagnostic messages to execute the bootloader.

While the microcontroller runs on a 32-bit architecture, both addresses required as arguments in the routine control preceding the download are only 3 bytes long. The ECU translates these by prepending them with `0xFF`, resulting in an address located in RAM. Before the ECU executes the downloaded code, it activates the watchdog timer in reset mode, which generates an unmaskable reset interrupt when the timer overflows, making the microcontroller reboot. The secondary bootloader can circumvent this mechanism by resetting the timer before an overflow occurs, implying that the unit will only resume its normal functionality once the bootloader performs a manual reset, for instance by jumping to the reset vector.

Trace 1. Executing the secondary bootloader on a Fiat BSI

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

5 Building a Firmware Modification and Extraction Framework

We demonstrate the capabilities of the secondary bootloader by developing a firmware modification and extraction framework. Using the procedure detailed in Sect. 4, we can execute arbitrary code on any ECU that implements this mechanism. The code downloaded to the ECU is binary machine code, so at the very least we must know the architecture of the ECU. Many microcontrollers used in ECUs are automotive-grade microcontrollers and thus incorporate at least

Table 3. ECUs on which we implemented the firmware extraction framework

Make	Year	Model	ECU	Microcontroller	Architecture
Ford	2012	Focus MK3	Body control module	MC9S12XEP768	HCS12X
	2012, 2014, 2016	Focus MK3	Instrument cluster	μPD70F3425	V850E
	2008	Fiesta MK6	Instrument cluster	MC9S12HZ256	HCS12
	2013, 2014, 2015, 2017	Fiesta MK7	Instrument cluster	MC9S12XEQ384	HCS12X
Volvo	2015	V50	Telematics unit	SH7267	SH2A[a]
Fiat	>2012	500	Body system interface	μPD70F3379	V850E1
	<2012	Grande Punto	Body system interface	μPD70F3237	V850E1

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

one on-chip CAN interface. This framework aims to transmit the firmware over CAN so the code must contain a minimal microcontroller-specific CAN driver with transmitting capabilities. Table 3 lists the ECUs on which we implemented this framework, along with the incorporated microcontroller and the architecture on which it runs. We built a cross toolchain from the GNU GCC source to compile our code for each architecture we encountered.

Downloading and Executing the Code. The ECU only accepts downloads to a specific area in RAM which varies in different ECUs. Additionally, some units only accept a *RequestDownload* message with a 4 byte address and a 4 byte size, while others are more flexible. UDS provides a set of common negative response codes. If the ECU receives a request with the incorrect format, it replies with a negative response with code `0x13`, which means *incorrect message length or invalid format*. Contrarily, if the format of the request is correct but the address or size is not within the correct range, the unit responds with error code `0x31`, indicating *request out of range*. The ECU does not limit the amount of unsuccessful download requests, so we can find this address by covering the complete address space of the microcontroller. Provided that we know the memory layout of the microcontroller, we can limit the range significantly since the address is located in RAM. To further reduce the range, we can increment the address by 0x10 each time while the size remains constant. With a common ECU RAM size of 128 KiB, that makes for a maximum of 8192 attempts.

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

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

6 Mitigation

The only security measure preventing an attacker from downloading code to the unit is the security access service. It is therefore crucial that the challenge-response protocol implemented by the manufacturer is cryptographically sound. Khan [15] proposes the use of the Advanced Encryption Standard for the challenge-response protocol. Given the keys are diversified per car and ECU this would enhance the seed-key security significantly. However, since AES is a symmetric key encryption scheme, the encryption key must be stored in the firmware of the ECU. Unless special hardware is used to protect against reading this encryption key, an attacker can recover the secret key and use it on other ECUs which employ the same key.

A public-key based approach would mitigate the key diversification issues and does not require additional hardware. When a diagnostic client is connected, no time constraints are in place since the car is meant to be stationary during diagnostic maintenance. To mitigate the risk of replay attacks, the challenge is 128 bits long. The diagnostic client generates the response by signing the received challenge with its private key. The ECU verifies the response under the public key, which can be stored in the firmware of the unit. With the computational limitations of ECUs in mind, often running on a 32-bit or even 16-bit architecture, the Elliptic Curve Digital Signature Algorithm (ECDSA) with curve NIST P-256 [21] would be a suitable candidate [8], resulting in a response length of 512 bits.

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

Finally, more secure CAN communication would mitigate the risk of an attacker controlling the complete network from a previously compromised node. Radu et al. proposed LeiA [22], a light-weight authentication protocol for ECUs connected to the CAN bus. In order to transmit on a certain CAN ID, a node must have the authentication key corresponding to that identifier. A node transmits a Message Authentication Code (MAC) along with each message. Receiving nodes can check the validity of the sender simply by computing the same MAC. In this scenario, a node would be secure against attacks from the internal network if no other node has the authentication key for its diagnostic CAN ID.

7 Discussion

Security of Diagnostic Authentication Mechanisms. All the ciphers studied in Sect. 3 use some form of proprietary cryptography, with an insufficient challenge and response size of 24 or 32 bits, and an equally small internal state of the cipher. We have shown that if an attacker can obtain a challenge-response pair they can then often recover secret keys of the cipher. No time constraints exist when the ECU is connected to a testbench, as described in [19], making a successfull attack over CAN possible.

Efficiently generating and diversifying cryptographic keys for each individual car and ECU remains a difficult issue to solve for manufacturers, as shown in previous research [7,28]. Valasek and Miller raised the issue of diagnostic key diversification when extracting a set of secrets from a diagnostic device. They (re)used these secrets to authenticate to two ECUs under test. We have encountered similar issues for diagnostic secrets. From our experiments, diagnostic secrets are not diversified for ECUs in each car. An attacker who can recover the secrets for one ECU often has access to other ECUs of the same type or function, since manufacturers reuse these across different models.

Implications. There are several implications of the insecurity of the bootloader mechanism. Firstly, by dumping the firmware of security sensitive ECUs (such as the Passive Keyless Entry or immobilizer), an attacker can recover cryptographic keys necessary to unlock or start the vehicle. An attentive reader might say that an attacker with access to the internal network does not need to recover cryptographic keys. However, Checkoway et al. present an analysis of remote attack services in [4]. More remote vulnerabilities are covered in the literature [6, 23, 24]. These are often generic to the model or even make of the car, implicating that if an attacker gains access to a car through one of these generic remote channels, they could read out cryptographic keys specific to that car.

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

Responsible Disclosure. Following standard responsible disclosure practise, we have informed the relevant car manufacturers of the vulnerabilities described in this paper in April 2018, five months ahead of publication. It should be noted that, even though the production of an ECU is outsourced to a third party (a Tier 2 or 3 supplier), the OEMs specify the required diagnostic functionality in their ECUs.

8 Conclusion

In this paper we expose several vulnerabilities in diagnostic security. Firstly, we demonstrate how an attacker can bypass the challenge-response security used in diagnostic protocols. All the studied ciphers use some sort of proprietary cryptography, namely a slighlty adapted version of the Galois-LFSR. 32- or 24-bit challenges and responses and an equally small internal state further add to the insecurity of the ciphers. We demonstrate this by conducting an attack over CAN and recovering secrets through a limited amount of challenge-response pairs. Furthermore, we document the secondary bootloader, a piece of machine code which a CAN node can download to the RAM of a connected ECU through various diagnostic functions. An attacker can abuse this mechanism to recover cryptographic keys, adjust variables in memory or simply disable the ECU. Utilising the functionality implemented for this secondary bootloader, we build a generic firmware modification and extraction framework. To conclude, the challenge-response protocol is the main (and often only) access control mechanism on the ECUs we have studied. The proprietary ciphers used in this protocol are substandard, making it possible for an attacker to bypass these and control all peripherals of the microcontroller through the secondary bootloader, which they can download to RAM. Well deployed public-key cryptographic primitives would mitigate both of these issues.

References

1. The Universal Measurement and Calibration Protocol Family. Standard, Association of Standardisation and Automation and Measuring Systems (2016)
2. Bogdanov, A.: Linear slide attacks on the KeeLoq block cipher. In: Pei, D., Yung, M., Lin, D., Wu, C. (eds.) Inscrypt 2007. LNCS, vol. 4990, pp. 66–80. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79499-8_7
3. Bono, S., Green, M., Stubblefield, A., Juels, A., Rubin, A.D., Szydlo, M.: Security analysis of a cryptographically-enabled RFID device. In: Proceedings of the 14th USENIX Security Symposium (USENIX Security 2005), pp. 1–16. USENIX Association (2005)
4. Checkoway, S., et al.: Comprehensive experimental analyses of automotive attack surfaces. In: 20th USENIX Security Symposium (USENIX Security 2011). USENIX Association (2011)
5. European 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 J. Eur. Communities L **350**(28), 12 (1998)
6. Foster, I., Prudhomme, A., Koscher, K., Savage, S.: Fast and vulnerable: a story of telematic failures. In: Proceedings of the 9th USENIX Conference on Offensive Technologies, WOOT 2015 (2015)
7. Garcia, F.D., Oswald, D., Kasper, T., Pavlidès, P.: Lock it and still lose it-on the (in) security of automotive remote keyless entry systems. In: 25th USENIX Security Symposium (USENIX Security 2016), pp. 929–944. USENIX Association (2016)

8. Gura, N., Patel, A., Wander, A., Eberle, H., Shantz, S.C.: Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 119–132. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28632-5_9
9. Indesteege, S., Keller, N., Dunkelman, O., Biham, E., Preneel, B.: A practical attack on KeeLoq. In: Smart, N. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 1–18. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78967-3_1
10. Road vehicles - controller area network (CAN) - part 1: data link layer and physical signalling. Standard, International Organization for Standardization, Geneva, CH (2015)
11. Road vehicles unified diagnostic services (UDS) specification and requirements. Standard, International Organization for Standardization, Geneva, CH (2006)
12. Road vehicles diagnostic systems keyword protocol 2000 part 3: application layer. Standard, International Organization for Standardization, Geneva, CH (1999)
13. Diagnostic Connector Equivalent to ISO/DIS 15031–3. Standard, SAE, International (2012)
14. Kasper, M., Kasper, T., Moradi, A., Paar, C.: Breaking KeeLoq in a flash: on extracting keys at lightning speed. In: Preneel, B. (ed.) AFRICACRYPT 2009. LNCS, vol. 5580, pp. 403–420. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02384-2_25
15. Khan, J.: ADvanced Encryption STAndard (ADESTA) for diagnostics over CAN. SAE Int. J. Passeng. Cars - Electron. Electr. Syst. **8**(2), 296–305 (2015)
16. Kleinknecht, H.: Can calibration protocol version 2.1. Germany: ASAM eV, pp. 2–18 (1999)
17. Koscher, K., et al.: Experimental security analysis of a modern automobile. In: 2010 IEEE Symposium on Security and Privacy (SP), pp. 447–462. Institute of Electrical and Electronics Engineers (2010)
18. Miller, C., Valasek, C.: Adventures in automotive networks and control units. Def. Con. **21**, 260–264 (2013)
19. Miller, C., Valasek, C.: Car hacking: for poories. Technical report, IOActive Report (2015)
20. Nolte, T., Hansson, H., Norström, C., Punnekkat, S.: Using bit-stuffing distributions in can analysis. In: IEEE Real-Time Embedded Systems Workshop at the Real-Time Systems Symposium (2001)
21. Pornin, T.: Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). RFC 6979 (2013)
22. Radu, A.-I., Garcia, F.D.: LeiA: a lightweight authenticatiton protocol for CAN. In: Askoxylakis, I., Ioannidis, S., Katsikas, S., Meadows, C. (eds.) ESORICS 2016. LNCS, vol. 9879, pp. 283–300. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45741-3_15
23. Rouf, I., et al.: Security and privacy vulnerabilities of in-car wireless networks: a tire pressure monitoring system case study. In: 19th USENIX Security Symposium (USENIX Security 2010). USENIX Association (2010)
24. Valasek, C., Miller, C.: Remote exploitation of an unaltered passenger vehicle. Technical report, Illmatics (2015)
25. Vector Informatik: Product Catalog 5 (2010)
26. Verdult, R., Garcia, F.D.: Cryptanalysis of the megamos crypto automotive immobilizer. USENIX; login, pp. 17–22 (2015)

27. Verdult, R., Garcia, F.D., Balasch, J.: Gone in 360 s: hijacking with Hitag2. In: 21st USENIX Security Symposium (USENIX Security 2012), pp. 237–252. USENIX Association (2012)
28. Verdult, R., Garcia, F.D., Ege, B.: Dismantling megamos crypto: wirelessly lock-picking a vehicle immobilizer. In: 22nd USENIX Security Symposium (USENIX Security 2013), pp. 703–718. USENIX Association (2013)