My other car is *your car*: compromising the Tesla Model X keyless entry system

Lennert Wouters, Benedikt Gierlichs and Bart Preneel

imec-COSIC, KU Leuven, Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
firstname.lastname@esat.kuleuven.be

Abstract. This paper documents a practical security evaluation of the Tesla Model X keyless entry system. In contrast to other works, the keyless entry system analysed in this paper employs secure symmetric-key and public-key cryptographic primitives implemented by a Common Criteria certified Secure Element. We document the internal workings of this system, covering the key fob, the body control module and the pairing protocol. Additionally, we detail our reverse engineering techniques and document several security issues. The identified issues in the key fob firmware update mechanism and the key fob pairing protocol allow us to bypass all of the cryptographic security measures put in place. To demonstrate the practical impact of our research we develop a fully remote Proof-of-Concept attack that allows to gain access to the vehicle's interior in a matter of minutes and pair a modified key fob, allowing to drive off. Our attack is not a relay attack, as our new key fob allows us to start the car anytime anywhere. Finally, we provide an analysis of the update performed by Tesla to mitigate our findings. Our work highlights how the increased complexity and connectivity of vehicular systems can result in a larger and easier to exploit attack surface.

Keywords: Remote Keyless Entry, Passive Keyless Entry and Start, automotive security, reverse engineering, Bluetooth Low Energy, Secure Element

1 Introduction

Over the years motor vehicles have been extended with electronic features that aim to provide a higher level of security. This move towards electronic security features was instigated by a need for better security, laws mandating the use of these features and added convenience for the end user. One infamous example is the Ford Sierra (Sapphire) Cosworth, introduced in 1986, which was so often stolen that it became difficult to get the car insured. Cases like these in combination with laws (e.g. EU Directive 95/56/EC [Com95]) resulted in the introduction of electronic security features such as the immobiliser system.

In a classical implementation the immobiliser is part of the ignition system which not only authenticates the physical shape of a key blade but also authenticates a transponder chip contained within the key fob. At first these transponder chips stored a static identifier and were later on replaced by transponder chips that implemented a cryptographic challenge-response protocol. The first such cryptographic transponder chip was introduced by Texas Instruments in 1995 under the name Digitial Signature Transponder [Kai08]. The first version of these cryptographic transponders relied on the proprietary DST40 cipher. The use of this cipher in both automotive immobiliser systems as well as the Exxon-Mobile Speedpass payment system attracted the attention of security researchers. In 2005 Bono et al. reverse-engineered the proprietary cipher using a black-box approach. As the cipher had a keyspace of only 40-bits they were able to brute force the cryptographic keys using a cluster of Field-Programmable Gate Arrays (FPGAs). Nevertheless, almost 15 years

later, the DST40 cipher was still used in vehicles made by Tesla, McLaren, Karma and Triumph [WMA+19]. Texas Instruments introduced a proprietary 80-bit variant of the cipher, named DST80, back in 2008. Wouters et al. reverse engineered and published the DST80 cipher and discovered several security weaknesses in DST80-based immobiliser systems [WVdHG+20]. Specifically, they found weak key diversification algorithms being used by Toyota, Kia and Hyundai as well as a downgrade vulnerability in the second generation of the Tesla Model S key fob. Additionally, they demonstrated the application of voltage fault injection to recover ECU firmware as well as side-channel analysis to recover cryptographic keys from DST transponders.

Researchers have demonstrated cryptanalytic flaws in many proprietary encryption algorithms used by the automotive industry [Bog07, IKD+08, VGB12, VGE13, HGO18, VVB18]. While most of these flaws resulted in practical attacks, researchers have also discovered several implementation weaknesses in systems using these proprietary ciphers. For example, researchers have shown that physical side-channel attacks can be used to recover the 64-bit KeeLoq key stored in a key fob or the 64-bit manufacturer key stored in the receiver [EKM+08, KKMP09]. The recovery of the master key in a KeeLoq system can result in a system-wide compromise as all other cryptographic keys are derived from it. Garcia et al. demonstrated that the remote keyless entry systems used by the Volkswagen group since 1995 relied on a small number of global master keys [GOKP16]. The fact that there are a few master keys makes it possible to clone the keyless entry portion of these key fobs by eavesdropping on a single transmission.

Since the introduction of electronic security features in motor vehicles we have seen an increased interest from the research community in assessing the security provided by these systems. However, the vast majority of research papers focuses on systems that rely on security through obscurity or that use proprietary ciphers instead of standardised cryptographic primitives. In contrast, in this work, we perform a thorough security analysis of a modern day passive keyless entry and start system relying on secure public-key and symmetric-key primitives implemented in a Common Criteria certified Secure Element. We provide a detailed description of all the involved components and demonstrate that the added complexity (mainly due to the use of complex microcontrollers implementing Bluetooth technology) of these next-generation key fobs can introduce new attack vectors. By exploiting these attack vectors we demonstrate that using a Common Criteria certified Secure Element is not sufficient to obtain a secure product. In fact, the demonstrated attack does not exploit any weakness in the Secure Element itself but rather the way it is used.

Finally, having carried out our attack, we analyse the software update performed by Tesla to mitigate the issues described in this paper. Even though the analysis performed in this work is specific to the Tesla Model X, the presented techniques and methodology can be useful for others evaluating automotive products or other hardware-based security systems.

Outline

In Sect. 2 we provide the reader with a high-level overview of the components of the Model X PKES system. Sect. 3 describes how the Bluetooth Low Energy interface on the Model X key fob can be abused by an attacker. The body control module's diagnostic interface is explored in Sect. 4. We detail the key fob pairing protocol and how we reverse engineered it in Sect. 5. The vulnerabilities identified in Sect. 3 and 5 are then combined in Sect. 6 to create a Proof-of-Concept attack that allows to unlock and start the car in a matter of minutes. Finally, in Sect. 7 we discuss the software update put in place by Tesla to mitigate our findings and conclude the paper.

2 System anatomy

For the purpose of this paper we can consider the key fob and the Body Control Module (BCM) [1] to be the main components involved in unlocking and starting the Tesla Model X. In practice other vehicle components are involved (e.g. the drive inverter), and the user is also able to unlock and start the car using a companion smart phone application.

The car will only perform actions that are requested by a key fob that was previously paired to the car. To pair a key fob to the car a service technician can use the Tesla Toolbox software. The key fob can be used to lock and unlock the car by the press of a button, this will be referred to as Remote Keyless Entry (RKE). Additionally, the Passive Keyless Entry and Start (PKES) functionality ensures that the car will automatically unlock and start if the key fob is in physical proximity. The goal of this section is to provide the reader with an overview of the main components.

2.1 The key fob

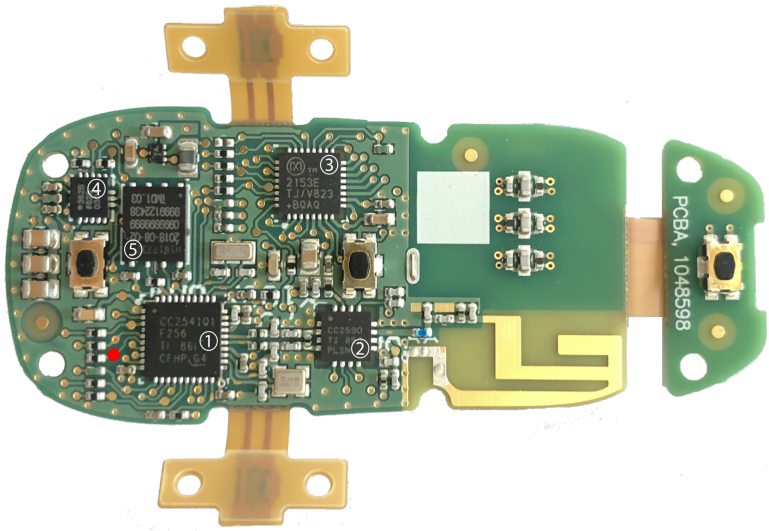

Figure 1: The Model X key fob PCB (top side). The main components are: (1) Texas Instruments CC2541 BLE SoC, (2) TI CC2590 BLE range extender, (3) Maxim Integrated MAX2153E 22 kHz transponder, (4) Analog Devices ADXL362 MEMS accelerometer and (5) Infineon SLM97CFX1M00PE Secure Element. The red circle indicates the test point for the Secure Element's IO interface.

Figure 1 shows the Printed Circuit Board (PCB) contained within the Model X key fob. Most notably, there is a Texas Instruments CC2541 BLE System-on-Chip (SoC) that is responsible for the key fob to car communication. Additionally, there is a Maxim Integrated Low Frequency (22 kHz) transponder chip that receives the data sent by the vehicle. Finally, there is also an Infineon SLM97 Secure Element that is Common Criteria EAL5+ certified. Interestingly, this key fob contains three microprocessors, each including a hardware AES co-processor.

Tesla opted for the less conventional choice of Bluetooth Low Energy (BLE) as the key fob to car communication channel. This choice was likely informed by a phone-as-key feature that would be able to use the same channel. Additionally, the LF communication channel at 22 kHz does not seem to be as widely used compared to the 125 kHz and 134.2 kHz channels. This less commonly used communication channel, combined with

[1] Tesla often refers to this component as the central body controller (BCCEN).

the accelerometer likely helps to prevent relay attacks. In a relay attack the maximum distance between the car and key fob in a passive entry scenario is extended, allowing to unlock and start a car even when the key fob is not in physical proximity [FDC11]. The uncommon communication channel is not compatible with most of the off-the-shelf relay tools. Additionally, the accelerometer can be used to put the key fob in a so-called deep-sleep mode when it is not moving: this increases battery lifetime and allows the key fob to ignore incoming communication attempts. When looking for car theft reports it is clear that several Tesla Model S vehicles have been stolen using relay attacks. However, at the time of writing, no such public reports exist about Tesla Model X vehicles. Nevertheless, in future versions it would make sense to explore secure distance bounding protocols to ensure the physical proximity of the key fob to the car. The secure ranging system developed by Abidin et al. seems particularly relevant as it was designed for narrow-band signals such as BLE [AESR+21].

The Model X key fob embeds lots of hardware and should contain all of the components needed to create a practically secure key fob. The use of a Common Criteria certified Secure Element leads us to believe that the key fob was designed with security in mind. By identifying the components used in the key fob it is clear that an attack will unlikely involve cryptanalysis of a proprietary cipher, instead the device is expected to use standardised cryptographic primitives. On the other hand, this key fob is more complex than classical designs, and has BLE as an additional attack vector. In Sect. 3 we demonstrate how the BLE interface can be abused by an adversary.

2.2 The Body Control Module

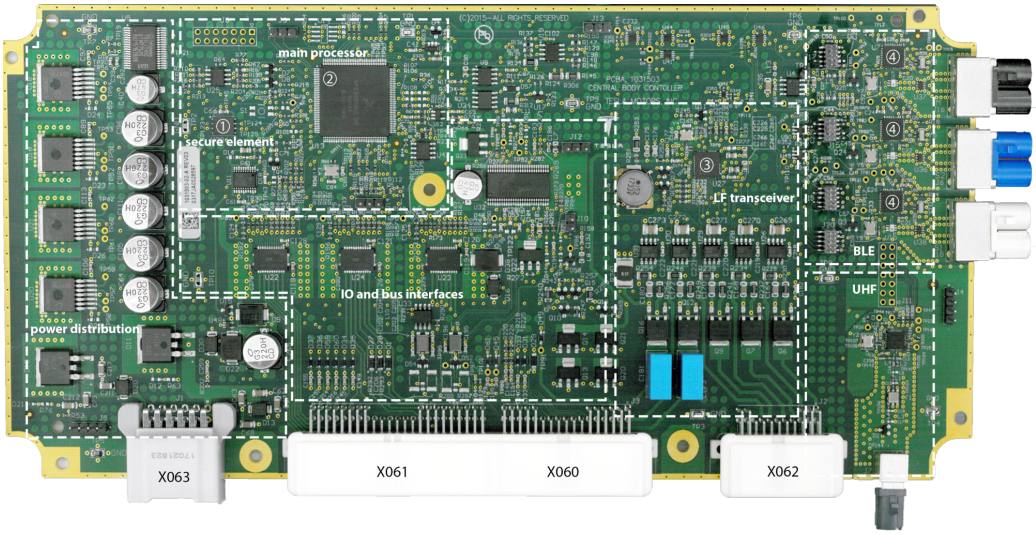

Figure 2: The Model X BCM PCB (top side). The main components are: (1) Infineon SLM97CFX1M00PE Secure Element, (2) Freescale SPC5605B, (3) Maxim Integrated MAX2143A 22 kHz transceiver and (4) Texas Instruments CC2541 BLE SoC.

The BCM in the Tesla Model X is, among other things, responsible for unlocking the doors, controlling interior lightning, and setting off the alarm siren. Figure 2 shows the PCB of the BCM and indicates the different zones of the module as well as the components relevant for this work. Both the key fob and the BCM use an Infineon SLM97 Secure Element. Unsurprisingly, the BCM contains a Maxim Integrated 22 kHz transceiver, enabling it to transmit LF signals using the five LF antennas distributed over

the vehicle. Additionally, the BCM contains three Texas Instruments CC2541 BLE chips (one for each BLE antenna) to receive data from the key fob.

The main processor in the BCM is a Freescale SPC5605 that communicates with the other components in the BCM, as well as other modules in the car. Communication with other components in the BCM happens using General Purpose Input and Output (GPIO) interfaces and inter-chip communication protocols such as UART and SPI. Communication with other ECUs happens primarily over the CAN and LIN interfaces available on connectors `X060` and `X061`. The most relevant CAN bus for this paper is referred to as the body CAN; it is available on the diagnostic connector (`X179`) located underneath the Media Control Unit (MCU) in the cabin.

The availability of these CAN busses on the diagnostic connector allows to interact with the ECUs in the vehicle (such as the BCM). This can be used by a service technician to read out Diagnostic Trouble Codes (DTC) and to pair a new key fob to the car. Most ECUs implement a Unified Diagnostic Services (UDS) interface that is accessible over the CAN bus. In Sect. 4 we describe how the UDS interface can be enumerated.

2.3 Diagnostic tools

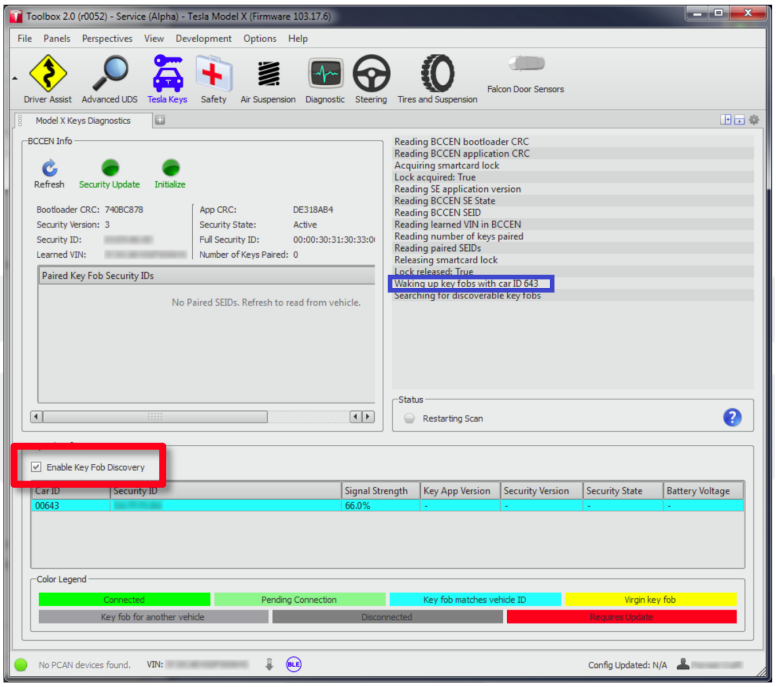

Figure 3: A screenshot of the Tesla Toolbox software. Note that this screenshot indicates that the tool can be used to wake up key fobs (blue rectangle) and that the tool can discover key fobs that are advertising (red rectangle). Picture source: https://static.nhtsa.gov/odi/tsbs/2016/SB-10081823-5448.pdf

The Tesla Toolbox software is a diagnostic tool used by service technicians for Model S and Model X vehicle maintenance. The Toolbox software also implements the key fob pairing routine and thus allows a service technician to pair a new key fob to the car. Under normal circumstances the software runs on a laptop with a USB-to-CAN interface that connects to the diagnostic connector in the vehicle. As mentioned earlier, this allows the software to communicate with the ECUs inside the vehicle. Figure 3 shows a publicly available screenshot of the Toolbox software. From this screenshot we can infer that the

Toolbox software can instruct the car to wake up a key fob and that the software can discover key fobs that are advertising and thus awake. In Sect. 3 we detail how one can interact with a key fob that is actively advertising and in Sect. 4 we demonstrate how a UDS routine can be used to wake up key fobs.

To pair a key fob an additional USB-to-BLE adapter is used, allowing the Toolbox software to communicate with the key fob. In a normal key fob pairing scenario the Toolbox software orchestrates the pairing process and effectively serves as communication bridge between the Secure Element in the key fob and the Secure Element in the BCM. The pairing protocol will be explained in more detail in Sect. 5. Many cars implement two key fob pairing scenarios, one in which a paired key fob is present and one in which there are no paired key fobs available. The former scenario often allows a car owner to pair an additional key fob to their car without the need to visit a service center. The latter is often referred to as an all keys lost scenario and usually requires manufacturer specific diagnostic tooling to resolve. For the Tesla Model X there is only one pairing mechanism and it does not require a paired key fob to be present. This unifies the pairing protocols but also prevents a legitimate owner to pair an additional key fob to the car without visiting a service center.

As with most automotive diagnostic tools the Tesla Toolbox is not available as a free download for anyone to use, but leaked versions do circulate on the internet. Depending on local regulation the software has to be made available to independent repair shops. In those locations a short-time subscription can be purchased from Tesla online. In our region a subscription that provides access to servicing information (including wiring diagrams) and the Toolbox software costs €125 for one hour of access, €175 on a daily basis, €750 on a monthly basis or €4500 on a yearly basis. Differences in regulation did make it briefly possible to download the software from the Tesla website by registering for a Chinese service account.[2] Tesla later disabled these newly registered accounts preventing them from logging in to and using the Toolbox software.

While we initially started this research project without access to the toolbox software we later obtained access to the module responsible for key fob pairing through an online Tesla reverse engineering community. Even though we could not use the Toolbox software in its intended manner it was still of great value when reverse engineering the key fob pairing protocol that is detailed in Sect. 5.

3 Key fob and its BLE inteface

As shown in Sect. 2.1 and Fig. 1, the Model X key fob uses a Texas Instruments CC2541 BLE SoC. During normal operation (RKE and PKES) the key fob does not advertise itself as being a connectable BLE peripheral. But it does use BLE advertisement packets to transmit data (e.g. a RKE unlock command) to the car. However, when power cycling the key fob it will briefly advertise itself as a connectable BLE peripheral. Similarly, the key fob can be forced to advertise by the BCM using a LF packet. When the key fob is advertising as connectable, a BLE central can connect to it and obtain a list of available services and their associated characteristics.

The Model X key fob hosts three BLE services. The first service contains characteristics that allow to read general information (e.g. the software version and battery level) from the key fob. Additionally, there is a service containing characteristics used for Over-Air-Download (OAD), Texas Instruments' implementation for Over-The-Air (OTA) firmware updates. In other words, this OAD service allows to update the firmware on the CC2541 BLE SoC wirelessly over BLE. The third service is related to the use of Application Protocol Datagram Units (APDUs): these are data units typically used to communicate with smart cards. In this case the service allows a BLE client to interact

[2] https://twitter.com/greentheonly/status/1334564079264993281

with the Secure Element inside the key fob, a feature used when pairing a new key fob to the car. The pairing process will be explained in more detail in Sect. 5.

Both the OAD and APDU services are interesting from an attacker's perspective. If the OAD firmware update mechanism is not properly secured, it can be abused by an attacker to upload a malicious firmware image to the CC2541 BLE SoC. The APDU service may be abused to send arbitrary APDU commands to the Secure Element in the key fob. In the remainder of this section we will take a look at these two services in more detail.

3.1 Secure Element interface

Secure Elements, such as the Infineon SLM97CFX1M00PE used in the Model X key fob, share similarities with smart cards. The physical smart card interface as well as the APDU format are standardised as part of the ISO-7816 specification [fS11]. In this case the physical interface is different because the key fob contains the SLM97 Secure Element in a PG-VQFN-8-1 package.[3] Nevertheless, the interface signals are the same as those used in a normal smart card (GND, VCC, IO, RST and CLK). For our purposes the Input Output (`IO`) signal is the most interesting, as it carries all of the information (APDUs) being exchanged between the CC2541 BLE SoC and the secure element.

Figure 1 shows where the `IO` signal can be accessed on the key fob PCB; all of the exchanged APDU commands can be received by connecting a logic analyser to this `IO` pin. Figure 4 shows a Model X key fob with wires soldered to most of the test points and uncovered vias on the PCB. Being able to receive multiple signals that carry information between the various components on the PCB helps to get an understanding of how the system functions. For example, by probing multiple signals at the same time we determined that a button press is received by the MAX2153E chip which then sends a signal over a Serial Peripheral Interface (SPI) to the CC2541 BLE SoC. Afterwards, the CC2541 BLE SoC sends an APDU command to the Secure Element which returns a 16-byte response. The APDU response is later broadcasted by the CC2541 and is a token instructing the vehicle to perform an action (e.g. lock or unlock).

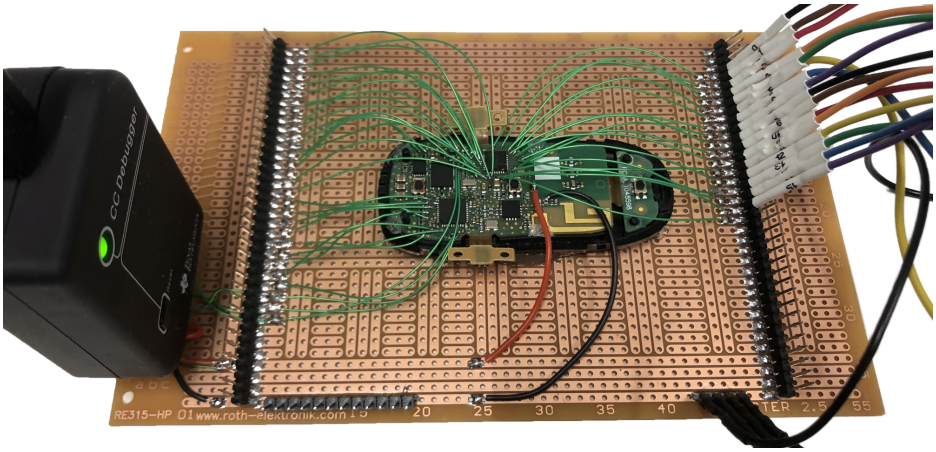

Figure 4: A Model X key fob prepared to provide easy access to inter-chip communication channels that can be analyzed by connecting a logic analyzer. Additionally, we added an interface that allows to program the TI CC2541 BLE SoC using a CC-Debugger.

As discussed earlier, the CC2541 provides an interface to the Secure Element allowing a BLE client to send APDU commands to the Secure Element through the APDU service.

[3] https://www.infineon.com/cms/en/product/packages/PG-VQFN/PG-VQFN-8-1

The APDU BLE service contains four main characteristics: `APDU command`, `APDU data`, `send APDU` and `APDU response`. Sending an APDU command to the Secure Element involves writing the main APDU command (typically five bytes) to the `APDU command` characteristic. Afterwards, additional APDU data can be written to the `APDU data` characteristic. Once the APDU command and APDU data are written, sending the actual APDU command to the Secure Element can be triggered by writing `0x01` to the `APDU send` characteristic. Through a notification the `APDU response` characteristic will signal when the APDU response can be read back from the `APDU data` characteristic.

By sending APDU commands through this BLE interface, and observing the response as well as the `IO` signal, it became clear that a blocklist is implemented in the CC2541 based on the APDU instruction field (INS). In other words, certain APDU commands, such as the one used after a button press, are blocked by the CC2541 when sent through the BLE interface. This blocklist was implemented to prevent an attacker from performing certain actions (e.g. requesting a valid unlock token) over the BLE interface.

3.2 OAD service

Texas Instruments provides two example implementations of OAD. The first implementation adds a Cyclic Redundancy Check (CRC) based integrity check to the firmware image. The second implementation aims to provide firmware confidentiality by encrypting it using AES in CounTeR (CTR) mode. Additionally, firmware authentication and integrity are provided based on AES-CBC-MAC.

In 2018 Seri et al. demonstrated a vulnerability in a customised implementation of OAD used in Aruba access points [SVZ18]. Aruba implemented an additional password check on top of the default (unprotected) OAD implementation. However, this password was hard-coded in the firmware and shared among all devices. While the example implementations provided by Texas Instruments have been around for many years, they still contained critical vulnerabilities in 2019 [Hof19]. For example, the AES-CTR implementation was flawed, resulting in a repetition of the keystream every 64 bytes. In most cases this would allow to decrypt the firmware image without any knowledge of the key. Additionally, the message authentication tag was checked using a non-constant time implementation of `memcmp`. These examples demonstrate that the OAD service on the Model X key fob can be an interesting target for attack, as a flaw in this service can allow an adversary to overwrite the firmware and obtain remote code execution.

The Toolbox software can be used to update the key fob's firmware and thus contains a copy of the latest firmware binary.[4] Additionally, the firmware for most components, including the key fob, in the car are contained within the root filesystem of the infotainment system as it is responsible for updating all components in the car. From a cursory analysis of the binary firmware update (version 1.5.1) it was clear that the firmware image is distributed in plaintext for the Model X key fob. The firmware image uses the same header format as the example implementation provided by Texas Instruments, but Tesla added some extra fields at the end of the firmware image. The firmware starts with a 16-byte header containing a 16-bit CRC value. The header is followed by the actual code portion of the firmware image and is padded with `0xFF` bytes to a fixed length. Finally, Tesla added the SHA1 hash of the firmware image followed by a 12-byte magic value consisting of two spaces, the text emoticon ¯\(°_°)/¯ and one additional space.

From this initial firmware format analysis we could not identify any signature or message authentication tag to protect the authenticity of the firmware. We verified this finding by modifying the device name (`Tesla Keyfob`) that is part of the BLE advertisements. Afterwards we updated the CRC value and SHA1 hash digest and performed the OAD protocol. The key fob now advertised with the device name we had set earlier and thus

[4] https://static.nhtsa.gov/odi/tsbs/2016/SB-10081823-5448.pdf

accepted the modified firmware. From this experiment it was clear that the key fob was not verifying the authenticity of the supplied firmware binary.

This insecure firmware update (or OAD) implementation allows an adversary to overwrite the firmware executed by the CC2541 SoC. In practice this means that an attacker who is able to establish a BLE connection will be able to execute arbitrary code on the keyfob's BLE SoC and thus send arbitrary APDU commands to the secure element. However, during normal operation the key fob will not advertise a connectable BLE peripheral. In Sect. 6.2 we demonstrate how the original firmware of the key fob can be modified to remove the blocklist on the APDU interface, enabling us to send arbitrary APDU commands to the Secure Element. In Sect. 7 we explain in more detail how this insecure firmware update mechanism may have made its way into the product.

4 The BCM and its UDS interface

As discussed in Sect. 2.2 the BCM in the Model X is connected to the CAN network exposed on the diagnostic connector. This diagnostic connector can be used by a service technician to interact with the BCM using UDS on CAN. UDS is a commonly used diagnostics communication protocol specified as part of the ISO-14229 standard [fS20]. Many ECUs implement a UDS server that allows clients to interact with it. The UDS server can implement multiple services that can be used by the client to perform diagnostic operations.

As most ECUs comply at least partially with the UDS standard, it is relatively straightforward to enumerate the available functionality. Nevertheless, identifying which of the services and their sub-functions allow to carry out a vehicle specific action can be a tedious endeavour. In this section we discuss the approach we used to enumerate UDS services and the routines implemented by the routine control service. Our goal for this enumeration phase was to identify the diagnostic operation used to instruct the BCM to transmit a wake-up packet to the key fob. Further, we wanted to find out how to communicate with the Secure Element in the BCM over the diagnostic connector.

To enumerate the BCM's UDS interface we created a bench setup consisting of a Model X BCM, a bench-top power supply and a USB-to-CAN interface. This allowed us to easily communicate with the BCM using Python and open-source tools that are part of the Linux CAN subsystem. Specifically we used socketCAN and the CAN-utils userspace tools as well as the can-isotp kernel module in combination with the python-can-isotp library [Lc21, Har21, Les21].

4.1 Enumerating UDS servers and services

In most cases the UDS servers on a CAN network can be identified by sending a UDS request to every possible value of the transmit arbitration ID (an 11-bit identifier) and observing the response. Specifically, we sent a UDS request to the `DiagnosticSessionControl` service that is part of the standard as it is often implemented. If a reply is received on the corresponding receiver arbitration ID (transmit arbitration ID `+ 0x10`) we assume that a UDS server is present on the identified address or arbitration ID. As only a single ECU is connected to our laboratory setup we can be certain that the identified UDS server is in fact the UDS server hosted by the BCM.

Having identified the UDS server address it is possible to enumerate the implemented services. This can be achieved by sending an empty UDS request to each service identifier and observing the response. If no response is received there is no service listening on the selected service identifier. As the ISO-14229 standard contains a list of services and their default service identifier, it is often straightforward to link the service identifier to the service's purpose.

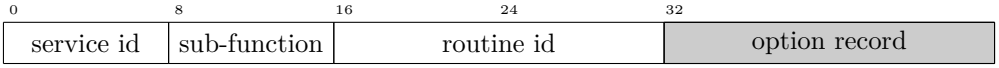

Figure 5: A UDS `RoutineControl` request message. The default service identifier for `RoutineControl` is `0x31`. Common sub-functions include start routine (`0x01`), stop routine (`0x02`) and request routine results (`0x03`). Some routines require additional input data (`routineControlOptionRecord`) that can be provided as part of the UDS message.

From this enumeration phase it was clear that the Model X BCM hosts multiple UDS services that allow to carry out various diagnostic actions. We made the assumption that the actions we wanted to perform (sending APDU commands to the Secure Element, and waking up a key fob) would be implemented as routines by the `RoutineControl` service. This assumption was confirmed by enumerating the UDS routines.

4.2 Enumerating UDS routines

The UDS `RoutineControl` service (service identifier `0x31`) allows a UDS client to start/stop routines and to request routine results. Figure 5 shows the structure of a `RoutineControl` request. Each request includes the service identifier, the command or sub-function that we want to perform and a two-byte routine identifier. Certain routines require extra input data referred to as the `routineControlOptionRecord` in the ISO-14229 specification.

The valid or existing routine identifiers can be enumerated by sending a UDS `RoutineControl` start request message for each routine identifier and observing the UDS response. For many of these routine identifiers the UDS service is likely to respond with a Negative Response Code (NRC). However, as these NRCs are defined as part of the UDS standard they can help guide further enumeration. For example, an NRC value of `0x33` corresponds to a `securityAccessDenied` error. This error indicates that the provided routine identifier is valid, but to use this routine we first have to authenticate to the UDS server using the `SecurityAccess` service. Similarly, an NRC with value of `0x13` corresponds to a `incorrectMessageLengthOrInvalidFormat` error. Such an error indicates that a routine with the provided routine identifier exists but that we did not provide the correct number of `routineControlOptionRecord` bytes. This information can be used to extend the enumeration process to determine the correct number of input bytes for each routine.

While enumerating the available routine identifiers is relatively straightforward, identifying what each routine does exactly is not. As mentioned earlier, the goal of our enumeration effort is to identify the routine responsible for sending APDU commands to the Secure Element and the routine that allows to send a wake-up command to the key fob. As a standard APDU command consists of at least five bytes we would expect the routine to require at least five `routineControlOptionRecord` bytes. In contrast, the routine to wake-up a key fob might not need any additional input beyond requesting the start sub-function. Our initial routine enumeration phase identified 54 routines, out of which 11 did not require any additional input and 10 required more than five bytes of `routineControlOptionRecord`.

To identify the routine responsible for waking up a key fob we attached a LF antenna to the BCM and placed a paired key fob nearby. Using a Python script we then sent routine start requests for each of the identified routine identifiers while scanning for BLE devices. In this way we were able to automatically identify the UDS routine responsible for waking up the key fob. By experimenting with multiple key fobs it became clear that the LF wake-up packet contains an identifier as it can only be used to wake up key fobs paired to the BCM. From reverse engineering the Toolbox software we learned that the Vehicle Identification Number (VIN) is used to derive a 2-byte car identifier that is also

stored in the key fob. This car identifier is part of the wake-up message, allowing key fobs with a different car identifier to simply ignore the wake-up request. In Sect. 6 we modify a BCM to have it transmit LF wake-up packets that match a VIN we provide.

Similarly, to identify the routine used to communicate with the Secure Element we attached a logic analyser to the Secure Element's IO line. Using a Python script we then sent routine start requests with the required number of `routineControlOptionRecord` bytes. When the logic analyser triggered and contained the data we had provided as the `routineControlOptionRecord`, we knew which routine identifier could be used to send APDU commands. Unsurprisingly, the Secure Element's response can be retrieved by using the routine's request result sub-function.

5 Pairing a key fob to the car

In normal circumstances pairing a key fob to the car requires the owner to schedule a service appointment. The service technician will connect a laptop, running the Tesla Toolbox software, to the car through a USB to CAN interface. Additionally, a BLE connection will be established between the laptop and the new key fob that will be paired to the car.

The steps involved in the pairing process can be divided into two distinct parts or protocols: the new key fob is first provisioned and afterwards it is paired to the car. In practice both protocols are usually carried out successively by the service technician. In the next sections we provide a detailed description of both the provisioning and pairing protocols. Afterwards, we describe how we were able the reverse engineer the operations carried out in the Secure Element itself as well the issues that we identified in the protocol.

5.1 The provisioning protocol

During the first part of the pairing process, called provisioning, the Toolbox software will communicate with the Secure Element in the key fob over a BLE connection and with a Hardware Security Module (HSM) operated by Tesla over an internet connection. Figure 6 shows the parties involved in the provisioning protocol as well as the messages exchanged between them.

The Infineon SLM97 Secure Elements used in this system have five RSA slots. Each of these slots can store a 2048-bit RSA key pair as well as an associated certificate. In Fig. 6 we show that the first steps consist of loading two specific certificates (Tesla Root and Tesla APP) in the first two slots; these certificates are shared among all key fobs. In practice, new key fobs are sold with these certficates pre-loaded and the associated slots locked down to prevent them from being overwritten.

Afterwards the Toolbox software will request the Secure Element's unique identifier and provides the car's VIN number to the Secure Element. At this point the secure element is instructed to generate RSA keypairs for each of the remaining three slots. For each of these slots the Tesla Toolbox software will request the backend HSM to generate a certificate signed using the private key belonging to the Tesla Root CA (slot 0). Each certificate contains the car's VIN number, the secure element's unique identifier, the slot it was generated for and the public key. Presumably the idea behind this step is to ensure that, during the pairing protocol, the car would be able to verify the authenticity of the key fob that is being paired.

Finally, after all the certificates have been generated and loaded into the Secure Element, the three slots are also locked down, preventing the contents from being modified. After this step the Toolbox software will automatically commence with the pairing protocol.



Figure 6: The provisioning protocol as it is used during normal operation. The key fob Secure Element has five RSA slots, each slot can store a 2048-bit RSA key pair as well as an associated certificate. Once a slot is locked the contents can not be overwritten unless it is first unlocked using a signed APDU command generated by the backend HSM.

5.2 The pairing protocol

In the remainder of the key fob pairing process the Toolbox software acts as a central orchestrator and communication relay between the Secure Element in the body control module and the Secure Element in the key fob. The pairing protocol is depicted in Fig. 7 and will be elaborated upon.

The Toolbox software initiates the pairing protocol and forwards a 16-byte pairing challenge generated by the BCM Secure Element to the key fob Secure Element. The key fob SE generates a RSA-PSS-SHA256 signature over the challenge, SE ID and the public key from slot 3 using its private key from slot 2. The BCM SE verifies the signature and generates five 128-bit AES keys. Afterwards, the BCM SE appends a magic value to the AES keys and encrypts them using RSA-OAEP and the key fob's public key from slot 3. Afterwards, the key fob can decrypt these AES keys, and verify that the magic value is still present. The remainder of the pairing protocol consists of a few steps that are used to ensure that both parties have the same symmetric keys. To that end the BCM SE will encrypt one 128-bit block using AES-ECB, in which the plaintext exists of 8 (presumably) random bytes concatenated with a fixed string (`key_auth`). The ciphertext is sent back to the key fob SE that decrypts the pairing validation challenge and confirms that the fixed string (`key_auth`) is present in the plaintext. Finally, the key fob SE encrypts one 128-bit block using AES-ECB in which the plaintext consists of the random input generated by the BCM concatenated with its own random input; this token is again verified by the BCM.

If the verification succeeds the key fob SE will be in a paired state in which all of the cryptographic material inside the SE is locked and cannot be modified. There is an APDU command that supposedly allows a service technician to revert the SE state back to its initial state. However, this APDU command requires an input that is signed by the backend HSM and thus requires valid Toolbox credentials.

5.3 Reverse engineering SE internal operations

As mentioned before, the Toolbox software mainly functions as a communication bridge between the BCM SE and key fob SE. Even though reverse engineering the Toolbox software provided valuable information about the pairing protocol, it was not sufficient to understand the operations performed by the Secure Element.

Specifically, the Toolbox software revealed the APDU commands that were being used, and variable names provided some hints for the meaning of the data. To obtain a complete overview of the pairing protocol we thus had to reverse engineer the operations performed within the Secure Element. This was achieved in a black-box fashion by interfacing with a BCM SE and a key fob SE simultaneously using standard smart card readers, a custom PCB and Python scripts.

For example, from analysing the toolbox software it was clear that the pairing protocol would start with the generation of a pairing challenge by the BCM SE. This step could be easily replicated by sending the same APDU command to the BCM SE and observing the response. Afterwards, we could send this challenge to an unpaired key fob SE. By dissecting the key fob SE's response it became clear that it consisted of the BCM challenge, a SE identifier, the public keys from slot 2 and 3 and 256 bytes of unidentified information. From the obtained information we assumed that the unidentified data was in fact an RSA signature. This assumption was verified using a script that checks several combinations of input data and the RSA signature scheme.

Using this guess-and-determine approach we were able to recover all of the operations carried out by the Secure Elements.

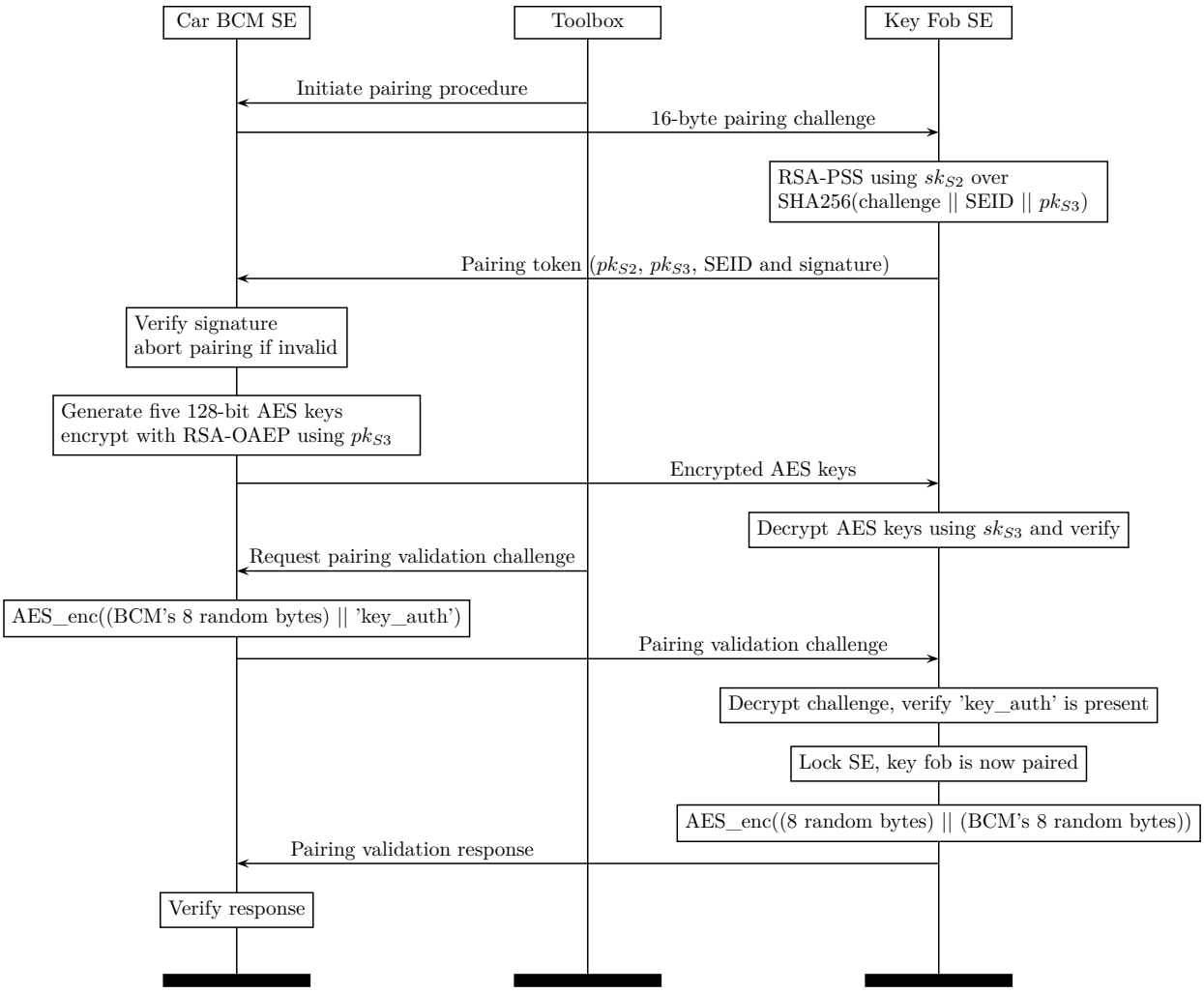

Figure 7: The pairing protocol as it is used during normal operation. We use || to denote the concatenation operation. pk_{Sx} and sk_{Sy} denote the RSA public key from slot x and the RSA private key from slot y respectively.

5.4 Identified issues

In 2017 Nemec et al. demonstrated that certain Infineon products generated RSA key pairs vulnerable to an extended version of the Coppersmith attack [NSS+17]. We retrieved the public keys from several Tesla Model X keyfobs (including some that contained secure elements produced in 2016) but were unable to find any vulnerable to this attack. It is unclear if this particular Infineon product was never vulnerable to the attack by Nemec et al. or if all of the tested key fobs received an applet update.

Anyway, the protocol itself contains a far more obvious weakness. Under normal circumstances a new key fob has to be provisioned (Sect. 5.1) before it can be paired (Sect. 5.2) to the car. Therefore, in practice, the provisioning and pairing steps are often performed successively by the same service technician. However, having gained a complete understanding of how these protocols work, it becomes obvious that the provisioning protocol can be skipped. The certificates generated and stored in the key fob's secure element during provisioning are never sent to the car during pairing. Therefore, it would be impossible for the car to verify the certificates and thus the authenticity of the key fob being paired. In Sect. 6 we demonstrate how the secure element in a key fob can be replaced, allowing to skip the provisioning protocol. This allows us to pair a malicious key fob to the car without requiring a valid service technician account.

6 Creating a Proof-of-Concept attack

The security issues outlined in this paper can be combined into a practical attack by executing the following steps:

1. Remotely wake up key fobs that are paired to the target vehicle:
   - The LF wake-up packet can be sent by a BCM (Sect. 4.2);
   - The packet contains an identifier based on the VIN which can be read from the windshield.
2. Connect to the target key fob using BLE and perform a firmware update (Sect. 3.2):
   - The malicious update disables the block list implemented on the APDU service (Section 3.1).
3. Use the APDU service to request a RKE unlock token from the secure element.
4. Use the acquired unlock token to unlock the car.
5. Connect to the diagnostic connector and pair a key fob to the car (Sect. 5.4).
6. The adversary now has a key fob that can be used to unlock and start the car just like an original key fob.

To demonstrate the practical applicability of our findings we implement a portable Proof-of-Concept device that can carry out the above steps. We favoured reusing and modifying Tesla components over custom hardware. While a custom design might result in a cheaper bill of materials, a smaller device or longer range, it would have required additional development time and reverse engineering. Nevertheless, the PoC device shown in Fig. 8 fits in a backpack and can be built from widely available components worth about USD 250.

In more detail, as an attacker we first have to wake up a target key fob such that it advertises as a connectable BLE peripheral. To do so we have to transmit a LF wake-up packet containing a car identifier that is derived from the VIN. However, the VIN is public

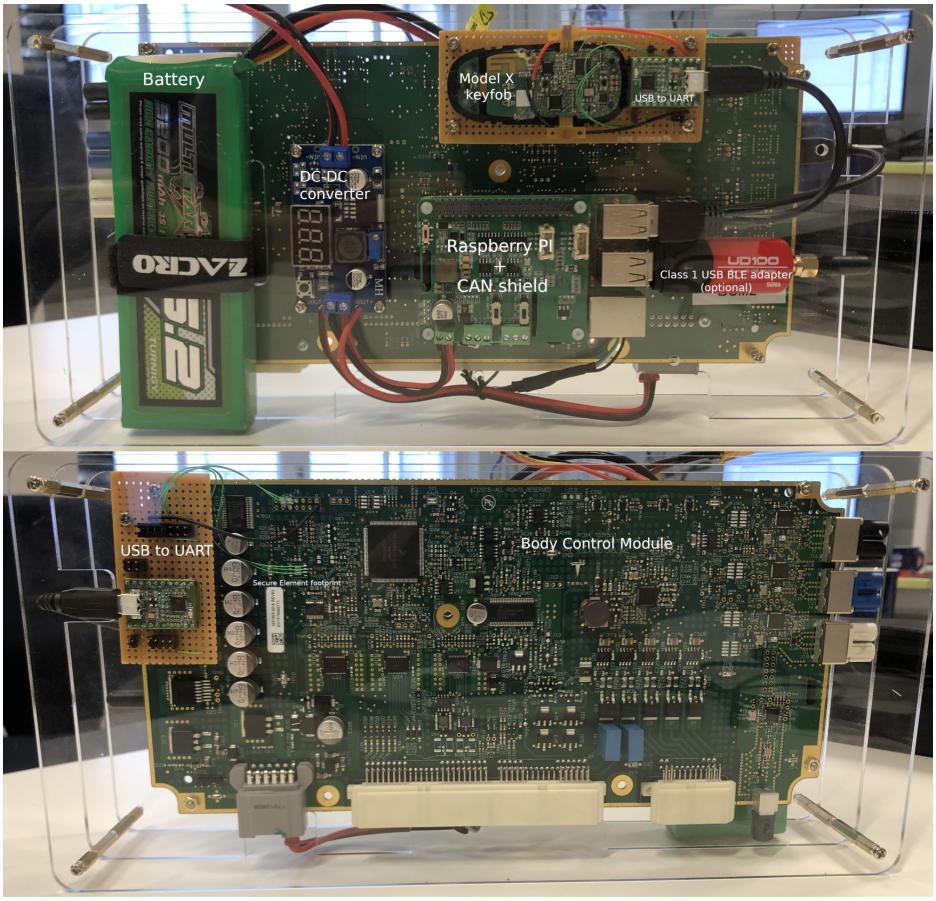

Figure 8: The PoC device consists of a battery and a DC-DC converter that is used to power the Raspberry Pi Model 3b+ with a 2-Channel CAN-BUS shield and the modified BCM.

information as it can be read from the windshield on the driver's side. This LF wake-up packet can be sent with a modified BCM by starting the wake-up UDS routine as discussed in Sect. 4.2. A key fob can be woken up using the BCM and the standard antenna used in the Model X over a range of a few meters. Once the key fob is advertising as connectable we can connect to it and push a malicious firmware update. The update process itself takes roughly 1.5 minutes but can be performed over a longer distance (BLE range). Once the malicious firmware update is completed we can request a valid RKE token from the secure element using the APDU service. Next, we can use this RKE token to unlock the car and gain access to the vehicles diagnostic connector located underneath the central display. We can then connect our device to this diagnostic interface to orchestrate the pairing protocol between the target vehicle and a modified key fob. Once paired to the car we can use our key fob to unlock and start the car. Additional security features such as pin-to-drive (disabled by default) would not prevent an attacker from creating a key fob that allows permanent physical access to the car but would prevent an attacker from driving off with the car.

The remainder of this section will describe some of the implementation aspects in more detail. Section 6.1 details how we modified a BCM and a key fob to be used in our PoC device. Section 6.2 describes how we modified the default firmware image to remove the blocklist implemented on the APDU service.

6.1 Simulating Secure Elements

In order to implement a full PoC attack our device requires the ability to wake up a target key fob remotely. We chose to modify a BCM for this purpose: as the wake up command transmitted by the BCM is derived from the VIN number, we required a way of changing the VIN number in the BCM. Similarly, as explained in Sect. 5, we require a key fob in which the secure element does not enforce a provisioning step.

Instead of making our own custom hardware to perform these tasks we abuse the implicit trust placed in the secure element by the other components in the key fob and the BCM. The CC2541 in the key fob interacts with the secure element through a Universal Asynchronous Receiver-Transmitter (UART) peripheral. Similarly, the SPC56 in the BCM communicates with the secure element using one of its UART peripherals. In both devices we can remove the secure element from the board and simulate its behaviour using a USB-to-UART bridge. Specifically, we used the Silicon Labs CP2102N USB-to-UART bridge as it supports non-standard baud rates (a baudrate of 21500 is required). Additionally, the CP2102N comes with additional GPIO pins that can be used to detect incoming RST signals for the Answer To Reset (ATR).

For both the BCM and the key fob we implemented the required secure element functionality in a Python script running on a Raspberry Pi with both the USB-to-UART peripherals attached to it. The modified BCM is used in the first step of our attack to wake up key fobs that are paired to a user configurable VIN number. The modified key fob can be paired to a car without provisioning and will thus be used in the fifth step of the attack. As the Secure Element in the key fob is being simulated, this same key fob can be used indefinitely and paired to multiple vehicles.

6.2 Brute-force firmware modification

As discussed earlier in Sect. 3.1, the APDU service exposed on the CC2541 BLE chip allows to send APDU commands to the secure element contained within the key fob. However, the APDU service implements a block list, a list of APDU instructions that are not usable through the BLE APDU service. We also established that it is possible for us to overwrite the stock firmware through the over-air-download service implemented on the CC2541 chip (Sect. 3.2). It is thus possible to push a custom firmware image to the key fob to obtain unrestricted access to the secure element.

It would be possible to achieve this goal by writing a custom firmware image from scratch, only implementing the required features to carry out an attack. This would require familiarity with the development tools and the target platform, and would likely require considerable effort. Another option is to use the stock firmware and slightly modify it to remove the APDU block list. As we do not have access to the source code of the stock firmware this requires modifying the compiled binary code. We opted for the second approach (binary modification), as this would not require us to learn the development environment. Additionally, this method results in a malicious firmware image that retains all of the functionality present in the stock firmware.

Locating the offset in the binary to patch can be achieved by reverse engineering the firmware image and identifying where exactly the block list is implemented. While most static reverse engineering tools do support the underlying 8051 instruction set, they do not implement support for the custom additions implemented in the CC2541. Static analysis of the binary firmware can be a time consuming effort, instead we opted for an automated approach.

The 8-bit 8051 instruction set lends itself well to a brute-force based firmware modification approach. The most straightforward way to implement a (short) block list would be a set of conditional if statements. These if statements are compiled into a set of instructions that likely contain the literal we are comparing to and a conditional jump. The

idea behind brute-force firmware modification is thus to replace potentially interesting occurrences of a certain byte and see if we end up with the desired effect.

The approach taken in this case works as follows. Modify an occurrence of the Jump if Accumulator Not Zero (JNZ) instruction (0x70) in the firmware to a Jump if Accumulator Zero (JZ) instruction (0x60), flash the modified firmware to a key fob using a CC-debugger (we used the setup shown in Fig. 4), connect to the keyfob over BLE and send the APDU command. If a response is received the modification successfully bypassed the block list, otherwise continue to the next occurrence of a JNZ instruction. This process can be easily automated and finds a solution within hours.

Once the offset which allowed to send the target APDU command was located, we were able to disassemble the surrounding instructions implementing the block list. Appendix A shows the 8051 Assembly instructions implementing the block list and the same portion of code with the blocklist disabled. Having generated our malicious firmware image we then updated the CRC and SHA1 hash to obtain a valid firmware binary that can be used with OAD in the second step of our attack. In the third step, this malicious firmware allows to read a valid RKE token from the secure element using the now unfiltered APDU service. This token can be transmitted to the car as a BLE advertisement packet to unlock it.

7 Discussion and conclusions

In this last section we discuss the aftermath of our findings and finally conclude our work.

7.1 Responsible disclosure

Back in August of 2020 we reached out to the Tesla product security team to disclose our findings. They were able to confirm the reported issues within a matter of days. As part of the 2020.48 update that rolled out in November of 2020, a firmware update was automatically pushed to the key fob while driving, without requiring any user interaction. We coordinated with the product security team to release some initial research findings at the same time as the update roll out. The details, covered in this paper, were not made public immediately to prevent the issue from being abused. Tesla awarded our research with a USD 5000 bounty through their bug bounty program.

For a user it is difficult to tell if the key fob has been updated or not, as the car does not provide any feedback regarding the update process. Because we received questions on how to verify that the update succeeded, we provide instructions on how to manually check the software version on a Model X key fob in Appendix B.

7.2 The mitigation update and root cause

As mentioned earlier, Tesla released a firmware update for the key fob to fix the OAD issue identified in Sect. 3.2. By comparing this new firmware image to the previous version we noticed that an extra 256-byte chunk of random looking data was present. We correctly assumed that this was a firmware signature. However, this observation was surprising because there would be no need to sign the new firmware image as the currently deployed firmware version did not seem to verify the signature of our malicious update.

To investigate further we pushed the new firmware image using the OAD protocol to one of our key fobs and confirmed that this new update was valid and accepted by the key fob. Additionally, we verified that the key fob no longer accepted our malicious firmware update. We also logged the data exchanged between the CC2541 BLE SoC and the Secure Element while performing the new firmware update. In doing so we were able to verify that the SHA-1 hash digest and the signature of the firmware image were being sent to the Secure Element for verification. We verified that the Secure Element uses

the Tesla Root public key (Secure Element slot 0) to verify the supplied SHA-1 hash and signature. Specifically, the Secure Element first computes the SHA-256 digest over the supplied SHA-1 digest and verifies the supplied signature using RSA-PSS. In Sect. 7.3 we discuss how the use of SHA-1 can lead to a practical attack in the near future.

Looking back at a key fob with the old firmware version and logging the Secure Element IO data it became clear that even for our malicious update the CC2541 BLE SoC was asking the Secure Element to verify the signature. Even though the Secure Element returned status words indicating a verification error, the CC2541 BLE SoC still accepted the firmware image. In other words, even the old firmware version was using the Secure Element to verify the authenticity of a new firmware update but was simply ignoring the result produced by the Secure Element. We can only speculate as to how this incomplete signature check ended up in the product. One possible explanation is that the signature check was partially disabled during development to speed up testing, but that the check was not enabled again in the production build.

7.3 The remaining attack surface

The firmware update performed by Tesla prevents us from easily overwriting the firmware on the CC2541. In doing so Tesla prevents us from removing the block list implemented on the APDU interface. However, the current firmware update solution that uses a SHA-1 digest can be prone to a chosen-prefix SHA-1 collision attack [LP20]. An attacker able to generate a useful firmware image with a SHA-1 digest that collides with the SHA-1 digest of a signed firmware image can recreate our attack.

Furthermore, as far as we know none of the other identified issues have been resolved. The pairing protocol remains unchanged and could thus still be abused by someone who gains physical access to the vehicle. For example, in an evil-maid, evil-spouse or evil-valet scenario an attacker could gain physical access, or with brief access to the key fob a valid RKE token can be pre-recorded. Similarly, to the best of our knowledge, the technique used to wake up a key fob can still be used at the time of writing. We believe that a significant reduction in attack surface can be achieved by disabling this functionality in paired key fobs, but it is not clear if this functionality has other purposes.

Fundamentally, the only thing that is required for someone to replicate our attack is another vulnerability in the CC2541 BLE SoC: once an attacker gets remote code execution on this chip over the wireless interface, they can again request valid RKE tokens from the Secure Element. This could be achieved if another logical or memory corruption vulnerability is identified in the BLE services implemented by Tesla or even in the BLE stack provided by Texas Instruments. Researchers have previously shown that such vulnerabilities exist and can be exploited [SVZ18, GWC+20]. This demonstrates the importance of being able to update such embedded devices in real world deployments.

Our work demonstrates that one fundamental weakness in this system is the way in which the Secure Element is used. As an attacker you do not have to compromise the Secure Element, but merely the chip that is used to interface with it. This is arguably a design flaw, and one that is difficult to resolve with the current hardware. The secure element cannot attest the authenticity of the software running on the CC2541. Any software based solution implemented on the CC2541 would likely be bypassable as it does not have a hardware root of trust. From a system level it would theoretically be possible to only use the BLE chip as a communication relay and to have both secure elements mutually authenticate before the car unlocks. However, such functionality would require two-way communication, usually not implemented in a RKE scenario. The use of Time-based One-Time Passwords for the unlock token instead of what effectively are Event-based One-Time Passwords would make our attack more difficult to execute, but such an implementation would require timekeeping ability.

Another attack vector that went largely unexplored in our work is the BLE interface in

the BCM: we do provide some notes for the interested reader in Appendix C. Similarly to the scenario described above, if a vulnerability is identified in the CC2541 chip a remote attacker might be able to gain control over it. However, in such a scenario the attacker would likely still have to pivot to the Freescale SPC5605B over the SPI interface before being able to unlock the car and potentially inject traffic on the CAN bus.

7.4 Conclusions

We provide a practical security evaluation of the Tesla Model X keyless entry system. In contrast to other keyless entry systems that have been analysed in the past, this system relies on secure cryptographic primitives. Besides covering the reverse engineering techniques, our analysis revealed two critical vulnerabilities that can be combined into a practical attack. The presented PoC implementation allows to steal a Tesla Model X in a matter of minutes. We abuse a first vulnerability to load a malicious firmware image to the key fob over its wireless BLE interface. In doing so we can instruct the Secure Element contained within the key fob to generate a valid RKE unlock token. This token can then be used to unlock the car. Having gained access to the car we can exploit a second vulnerability in the pairing protocol and pair a modified key fob to the car that allows to unlock and start the car.

Finally, we provide an analysis of the update performed by Tesla to mitigate our findings. Our work highlights how the increased complexity and connectivity of vehicular systems can result in a larger and easier to exploit attack surface.

Acknowledgments

We would like to thank the Tesla-REV community for providing valuable information and insight, and Ulrich for allowing us to use his car for testing. This work was supported in part by CyberSecurity Research Flanders with reference number VR20192203, in part by the Research Council KU Leuven C1 on Security and Privacy for Cyber-Physical Systems and the Internet of Things with contract number C16/15/058, and in part by the European Commission through the Horizon 2020 research and innovation programme under grant agreement Cathedral ERC Advanced Grant 695305.

References

[AESR+21] Aysajan Abidin, Mohieddine El Soussi, Jac Romme, Pepijn Boer, Dave Singelée, and Christian Bachmann. Secure, accurate, and practical narrowband ranging system. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):106–135, Feb. 2021.

[Bog07] Andrey Bogdanov. Linear Slide Attacks on the KeeLoq Block Cipher. In Dingyi Pei, Moti Yung, Dongdai Lin, and Chuankun Wu, editors, *Information Security and Cryptology, Third SKLOIS Conference, Inscrypt 2007, Xining, China, August 31 - September 5, 2007, Revised Selected Papers*, volume 4990 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2007.

[Com95] European Commission. Commission Directive 95/56/EC, Euratom of 8 November 1995 adapting to technical progress Council Directive 74/61/EEC relating to devices to prevent the unauthorized use of motor vehicles. 1995.

[EKM+08] Thomas Eisenbarth, Timo Kasper, Amir Moradi, Christof Paar, Mahmoud Salmasizadeh, and Mohammad T. Manzuri Shalmani. On the Power of

Power Analysis in the Real World: A Complete Break of the KeeLoqCode Hopping Scheme. In David A. Wagner, editor, *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, volume 5157 of *Lecture Notes in Computer Science*, pages 203–220. Springer, 2008.

[FDC11] Aurélien Francillon, Boris Danev, and Srdjan Capkun. Relay Attacks on Passive Keyless Entry and Start Systems in Modern Cars. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011.

[fS11] International Organization for Standardization. *Identification cards Integrated circuit cards: Part 1: Cards with contacts - Physical characteristics*. International Organization for Standardization, Vernier, Geneva, Switzerland, ISO/IEC 7816-1:2011 edition, 2011.

[fS20] International Organization for Standardization. *Road vehicles - Unified diagnostic services (UDS): Part 1: Application layer*. International Organization for Standardization, Vernier, Geneva, Switzerland, ISO 14229-1:2020 edition, 2020.

[GOKP16] Flavio D. Garcia, David F. Oswald, Timo Kasper, and Pierre Pavlidès. Lock It and Still Lose It - on the (In)Security of Automotive Remote Keyless Entry Systems. In Thorsten Holz and Stefan Savage, editors, *Proceedings of the 25th USENIX Security Symposium, Austin, TX, USA, August 10-12, 2016*. USENIX Association, 2016.

[GWC+20] Matheus E. Garbelini, Chundong Wang, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. SweynTooth: Unleashing Mayhem over Bluetooth Low Energy. In Ada Gavrilovska and Erez Zadok, editors, *Proceedings of the 29th USENIX Security Symposium, August 12-14, 2020*, pages 911–925. USENIX Association, 2020.

[Har21] Oliver Hartkopp. can-isotp. https://github.com/hartkopp/can-isotp, 2021.

[HGO18] Christopher Hicks, Flavio D. Garcia, and David Oswald. Dismantling the aut64 automotive cipher. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2):46–69, May 2018.

[Hof19] Robbert Hofman. Security evaluation of texas instruments cc254x bluetooth low energy chips. KU Leuven Master's Thesis, 2019.

[IKD+08] Sebastiaan Indesteege, Nathan Keller, Orr Dunkelman, Eli Biham, and Bart Preneel. A Practical Attack on KeeLoq. In Nigel P. Smart, editor, *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, volume 4965 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2008.

[Kai08] Ulrich Kaiser. Digital Signature Transponder. In Paris Kitsos and Yan Zhang, editors, *RFID Security: Techniques, Protocols and System-on-Chip Design*, pages 177–189. Springer US, Boston, MA, 2008.

[KKMP09] Markus Kasper, Timo Kasper, Amir Moradi, and Christof Paar. Breaking KeeLoq in a Flash: On Extracting Keys at Lightning Speed. In Bart Preneel, editor, *Progress in Cryptology - AFRICACRYPT 2009, Second International Conference on Cryptology in Africa, Gammarth, Tunisia, June 21-25, 2009. Proceedings*, volume 5580 of *Lecture Notes in Computer Science*, pages 403–420. Springer, 2009.

[Lc21] Linux-can. can-utils. https://github.com/linux-can/can-utils, 2021.

[Les21] Pier-Yves Lessard. python-can-isotp. https://github.com/pylessard/python-can-isotp, 2021.

[LP20] Gaëtan Leurent and Thomas Peyrin. SHA-1 is a Shambles: First Chosen-Prefix Collision on SHA-1 and Application to the PGP Web of Trust. In Srdjan Capkun and Franziska Roesner, editors, *Proceedings of the 29th USENIX Security Symposium, August 12-14, 2020*, pages 1839–1856. USENIX Association, 2020.

[NSS+17] Matús Nemec, Marek Sýs, Petr Svenda, Dusan Klinec, and Vashek Matyas. The return of coppersmith's attack: Practical factorization of widely used RSA moduli. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1631–1648. ACM, 2017.

[SVZ18] Ben Seri, Gregory Vishnepolsky, and Dor Zusman. BLEEDINGBIT: The Hidden Attack Surface Within BLE Chips. https://info.armis.com/rs/645-PDC-047/images/Armis-BLEEDINGBIT-Technical-White-Paper-WP.pdf, 2018. [Online; accessed 12-April-2021].

[VGB12] Roel Verdult, Flavio D. Garcia, and Josep Balasch. Gone in 360 Seconds: Hijacking with Hitag2. In Tadayoshi Kohno, editor, *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 237–252. USENIX Association, 2012.

[VGE13] Roel Verdult, Flavio D. Garcia, and Baris Ege. Dismantling Megamos Crypto: Wirelessly Lockpicking a Vehicle Immobilizer. In Samuel T. King, editor, *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 703–718. USENIX Association, 2013.

[VVB18] Aram Verstegen, Roel Verdult, and Wouter Bokslag. Hitag 2 hell - brutally optimizing guess-and-determine attacks. In Christian Rossow and Yves Younan, editors, *12th USENIX Workshop on Offensive Technologies, WOOT 2018, Baltimore, MD, USA, August 13-14, 2018*. USENIX Association, 2018.

[WMA+19] Lennert Wouters, Eduard Marin, Tomer Ashur, Benedikt Gierlichs, and Bart Preneel. Fast, Furious and Insecure: Passive Keyless Entry and Start Systems in Modern Supercars. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):66–85, May 2019.

[WVdHG+20] Lennert Wouters, Jan Van den Herrewegen, Flavio D. Garcia, David Oswald, Benedikt Gierlichs, and Bart Preneel. Dismantling DST80-based Immobiliser Systems. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(2):99–127, Mar. 2020.

Appendices

A Bypassing the blocklist

Listing 1 contains the 8051 assembly instructions that implement the APDU blocklist. The JNZ instruction marked in red was identified as the target instruction in our brute force firmware modification approach. In Listing 2 the JNZ instruction is replaced with a Short JuMP (SJMP) instruction. This modification ensures that the code will always jump to address `0x7a37` regardless of the flags set by the previous instruction. The entire block list can be bypassed by modifying all JNZ instruction in Listing 1 with SJMP instructions.

Listing 1: original blocklist

```
7a2f:   MOVX    A,@DPTR
7a30:   ADD     A,#0xd8
7a32:   JNZ     0x7a37
7a34:   LJMP    0x7b7f
7a37:   DEC     A
7a38:   JNZ     0x7a3d
7a3a:   LJMP    0x7b7f
7a3d:   ADD     A,#0xfc
7a3f:   JNZ     0x7a44
7a41:   LJMP    0x7b7f
7a44:   ADD     A,#0xfc
7a46:   JNZ     0x7a4b
7a48:   LJMP    0x7b7f
7a4b:   DEC     A
7a4c:   JNZ     0x7a51
7a4e:   LJMP    0x7b7f
7a51:   DEC     A
7a52:   JNZ     0x7a57
7a54:   LJMP    0x7b7f
7a57:   DEC     A
7a58:   JNZ     0x7a5d
7a5a:   LJMP    0x7b7f
7a5d:   ADD     A,#0xf8
7a5f:   JNZ     0x7a64
7a61:   LJMP    0x7b7f
```

Listing 2: patched blocklist

```
7a2f:   MOVX    A,@DPTR
7a30:   ADD     A,#0xd8
7a32:   SJMP    0x7a37
7a34:   LJMP    0x7b7f
7a37:   DEC     A
7a38:   JNZ     0x7a3d
7a3a:   LJMP    0x7b7f
7a3d:   ADD     A,#0xfc
7a3f:   JNZ     0x7a44
7a41:   LJMP    0x7b7f
7a44:   ADD     A,#0xfc
7a46:   JNZ     0x7a4b
7a48:   LJMP    0x7b7f
7a4b:   DEC     A
7a4c:   JNZ     0x7a51
7a4e:   LJMP    0x7b7f
7a51:   DEC     A
7a52:   JNZ     0x7a57
7a54:   LJMP    0x7b7f
7a57:   DEC     A
7a58:   JNZ     0x7a5d
7a5a:   LJMP    0x7b7f
7a5d:   ADD     A,#0xf8
7a5f:   JNZ     0x7a64
7a61:   LJMP    0x7b7f
```

B Reading the Model X key fob firmware version

The Tesla Model X does not provide an easy way to verify if the key fob has been updated. However, the key fob's firmware version can be read from one of its Bluetooth characteristics. To read the firmware version a smartphone app such as LightBlue or equivalent can be used. Before you can connect to the key fob, it first has to advertise itself as connectable. The easiest way to achieve this is to remove the battery from the key fob and reseat it. You should now be able to connect to the "Tesla Keyfob" using LightBlue. The firmware version can be read from the first characteristic (`0xF000FA01-`). If the returned value is `3E6A8D57A904A9E9` the key fob is running the old and vulnerable version of the firmware. At the time of writing, the latest key fob firmware version returns a value of `F1EF65AE0D7AF929`, and is no longer vulnerable to the OAD issue presented in this work.

C An inconclusive side-quest: arbitrary vehicle unlock

Even though this paper presents a practical attack and the techniques we used to achieve it, we believe it also interesting to share one of many endeavours that did not lead us to an exploitable path.

A RKE command is transmitted to the car by means of a BLE advertisement packet. This advertisement packet is received by the CC2541 BLE chips in the BCM and sent to the main processor (SPC5605B), if the packet structure is valid the main processor will send the token to the secure element for verification. If the verification succeeds the BCM will perform the requested action (e.g. unlocking the doors). The state of the door locks can be observed on the CAN bus and can thus be easily monitored, even in a basic bench setup.

Sniffing RKE advertisement packets (e.g. using `aioblescan`) is straightforward, by recording multiple advertisements the packet structure became evident. The data portion of the advertisement consists of `0x03` followed by a 2-byte car identifier, a 4-byte secure element identifier and a 16-byte token. Transmitting such advertisement packets can be easily achieved using e.g. `hcitool`.

As the Toolbox software contained artifacts of an early development version in which encrypted RKE commands were not yet implemented we decided to do some basic fuzzing of the advertisement packet format.

To that end we created a Python script that can send advertisement packets while monitoring the CAN bus for unexpected activity. We are aware that this is far from an optimal fuzzing setup, but it was sufficient for our purpose and took minimal time to setup. Unsurprisingly, we did not find a 'magic' advertisement packet that would unlock the car. However, we did manage to get the BCM into an unresponsive state in which the BCM would no longer unlock the doors, even if a valid RKE packet was sent using a key fob. While we were able to trigger this denial of service condition multiple times we were unable to reliably reproduce the issue.