Radboud University

Master Thesis

Secure updates in automotive systems

Author:
Remy Spaan - s4156889
remy.spaan@sogeti.com

Internal supervisor:
Peter Schwabe
peter@cryptojedi.org

Second reader:
Lejla Batina
lejla@cs.ru.nl

External supervisor:
Sjoerd Verheijden
sjoerd.verheijden@sogeti.com

May 30, 2016

Abstract

Modern cars are increasingly connected to the Internet, providing a variety of new features that are beneficial towards both drivers and car manufacturers. With all these new features comes more leisure, although it also introduces an entire new set of security issues. It is generally known among car manufacturers and security researchers that the current state of car security is weak. There are no real standards regarding car security on a physical or wireless level. In the recent years, several studies have been conducted on modern vehicles, with a security perspective in mind. This master thesis identifies the current shortcomings regarding automotive security by taking a closer look at these studies. Additionally, this thesis provides a model and a proof-of-concept implementation to secure a part of the update system of a widely used electronic control unit (ECU) in car systems. This proof-of-concept system provides aspects like confidentiality, authenticity and integrity of a supplied update, while preventing common security pitfalls. It uses implementations of cryptographic primitives designed for high speed and takes into account the constraints the ECUs are bound to. While this thesis does not cover all aspects of the update process, it takes a step towards the direction of making over-the-air firmware updates for car systems more secure.

Keywords. Car security, Over-the-air updates, automotive security, firmware updates, ECUs

Acknowledgements

With this thesis, an end of an era nears for me personally. The era of education. Elementary school, high school, college, and after a little break, university. That does not mean that I will stop learning about new things, hopefully. One is never too old to learn.

First and foremost I want to thank my father Jan for giving me the opportunity to entirely focus on my study the last few years. Without having to worry about a lot of things I could throw myself into this Master's education and without his support it would have been a very tough challenge.

I would also like to thank my university's supervisors Peter Schwabe and Lejla Batina for being my all-knowing *oracles* and giving me advice throughout the entire process from idea up to the result, this thesis.

Additionally, I want to thank everyone at Sogeti for giving me the opportunity to do my research at this company and keeping up with my awful word play jokes. My thanks go out to Sjoerd Verheijden, who kept me on track and kept me thinking of the bigger picture, without letting me drown in details. Also my thanks go to Koen ten Holter for providing me with the initial idea of this thesis and proofreading my thesis several times. Furthermore I would like to thank Rikkert ten Klooster for being my partner in crime (a bad choice of words with hacking in mind). Furthermore, I would like to give my thanks to Liis Jaks whose thesis was an inspiration to me and a cornerstone for this thesis. During my time at Sogeti, she was very knowledgeable and helpful on the subject and always had or made time to look at my thesis and discuss the outcome.

– Remy Spaan, May 2016.

Contents

List of Figures

List of Tables

List of abbreviations

ABS Anti-lock Breaking System

AES Advanced Encryption Standard

CAN Controller Area Network

CRC Cyclic Redundancy Check

DDoS Distributed Denial of Service

DoS Denial-of-Service

ECC Elliptic Curve Cryptography

ECDLP Elliptic Curve Discrete Logarithm Problem

ECU Electronic Control Unit

EEPROM Electrically Erasable Programmable Read Only Memory

FPGA Field-Programmable Gate Array

GPS Global Positioning System

HSM Hardware Security Module

IoT Internet of Things

LIN Local Interconnected Network

MAC Message Authentication Code

MITM Man In The Middle

MMU Memory Management Unit

MOST Media Oriented Systems Transport

NAT Network Address Translation

OBD On-Board Diagnostics

OEM Original Equipment Manufacturer

OTA Over-the-air

PCM Powertrain Control Module

PKI Public Key Infrastructure

PRNG Pseudorandom Number Generator

RCDLR Remote Control Door Lock Receiver

ROM Read-Only Memory

RSA Rivest Shamir Adleman

SCP Secure Copy Protocol

SHA Secure Hash Algorithm

SIM Subscriber Identity Module

SoC System on a Chip

SPE Signal Processing Extensions

SSH Secure Shell

SSL Secure Sockets Layer

TCU Telematics Control Unit

TLS Transport Layer Security

TRNG True Random Number Generator

VIN Vehicle Identification Number

1 Introduction

Automobiles are becoming increasingly computerized. Until recently, cars were controlled purely by mechanical means but now, thanks to digitalization, more cars have embedded computer systems to arrange control flows. While some of these flows are trivial, for instance door locking mechanisms, these same systems also control critical systems like brake pressure, engine power and airbags. Automobiles are controlled by multiple computers which are interconnected through wired and wireless systems. These Electronic Control Units (or ECUs) perform a great deal of actions that would normally not be possible, for instance the Anti-lock Breaking System (ABS) and remote car locking and unlocking. Without 70 to 100 of these embedded ECUs, modern cars would not be able to leave the driveway, because they contain firmware consisting of up to 100 million lines of code [1].

Due to the increasing amount of technology in car systems, these systems are becoming more vulnerable to attacks. Figure 1 illustrates the standard physical and wireless features of modern cars and their associated networks. Most of these features are supported by one or more ECUs. Regarding wireless security, the telematics component is the most relevant aspect. The Telematics Control Unit (TCU) of a car handles all incoming and outgoing data that is sent over the air. Think of a data connection for Internet access, GPS data, voice calls or connections to other systems by other means.

Recently, Tesla introduced over-the-air (OTA) firmware updates for its cars [2]. These firmware updates are distributed through the mobile network, as all of Tesla's car models have a Subscriber Identity Module (SIM) card included. It is estimated that by 2018, 36 million cars will have a SIM card embedded [3]. At first this embedded SIM card was only used for emergencies, like automatically dialing the emergency number if sensors detected a crash, but now companies are realizing that there are more possibilities. One of these possibilites is supplying car owners with over-the-air updates. This removes the need for customers to visit the garage to receive an update and enables manufacturers to update car systems on a more frequent basis. An over-the-air update could also unburden the manufacturer in case of a forced car recall due to an error or security flaw in the firmware.

Together with the introduction of all these new systems including the OTA update, a variety of potential attack vectors have surfaced. During an over-the-air update, the binary firmware passes through untrusted communication channels. First, a wireless channel is used to send the firmware to the car such as the cellular network, WiFi or Bluetooth. Then, physical channels inside the car, for instance a Controller Area Network (CAN) or Ethernet connection, distribute the firmware from the TCU to the correct ECU. Data passing these channels can be eavesdropped by an attacker and therefore, the firmware can be obtained or modified with potentially devas-

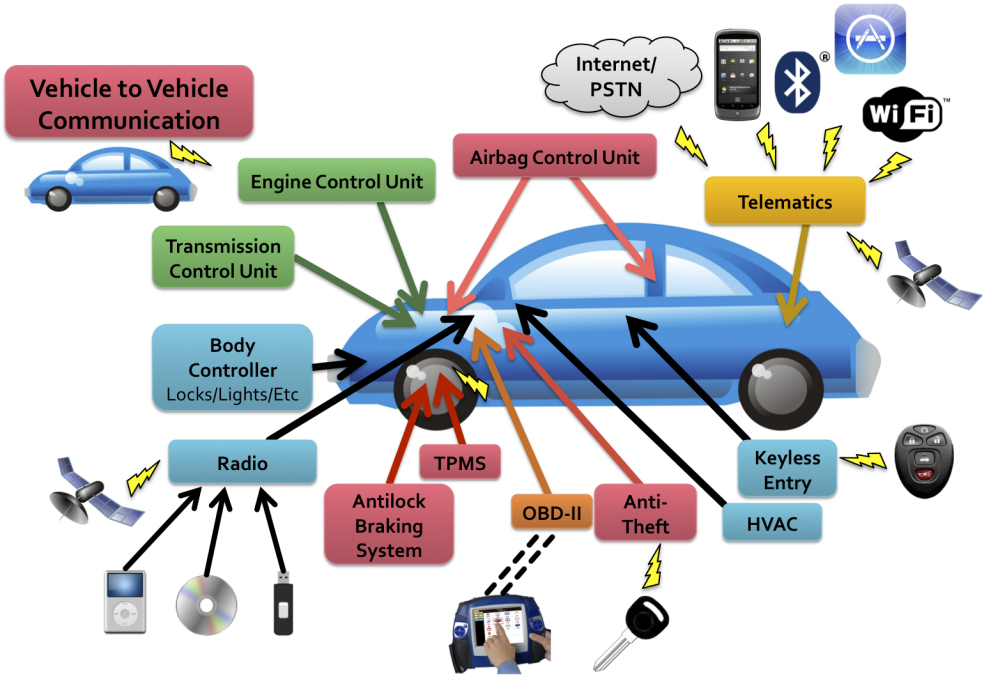

Figure 1: Car Networking. The different color nodes shows how the car features are usually grouped. HVAC stands for heating, ventilation and air conditioning, TPMS stands for tire-pressure monitoring system, OBD stands for on-board diagnostics. Source: [4].

tating effects. Hence, this update process should be protected from being tampered with or read by possibly malicious individuals or organizations such as car tuners, competing manufacturers or hackers.

At the moment there are no standards available regarding the required security of automobiles. There is an ISO 11898 standard [5] that specifies the physical and datalink layers of the CAN, which is one of the available transport mechanisms of data in a car, but this standard does not include any security aspects. Most implemented security features are proprietary and security has not been considered necessary for a long time. Security plays an increasingly bigger role in modern life so adequate security of systems we use inherently becomes more important. More devices are connected to the internet, and with the Internet of Things (IoT) taking a more prominent place in our daily lives, the security of these devices needs be to addressed too.

This thesis gives an insight into the current research towards the current state of physical and wireless security of a car. Additionally, it provides an update model independent of the wireless channel that is used and a proof-of-concept implementation of an ECU handling, authenticating and verifying a new firmware version. The proof of concept uses proven secure cryptographic primitives for encryption, authentication and verification of the firmware and is implemented on

a widely used microprocessor used in ECUs, the MPC5566.

This thesis is structured as follows: The next Section will describe some of the work and research that has been done on this subject in the past and which initiatives to improve car security are currently ongoing. Section 3 and 4 describe the attacker model and required security properties for our proof-of-concept implementation. Section 5 mentions the target platform and the to-be-used cryptography, an elaboration on the choice of the used primitives, which results in the used models in Section 6. Section 7 defines the variables, keys and functions that are used in our model while Section 8 focuses on the implementation itself. Ultimately, Section 9 contains the results and future work that might be interesting to investigate with the results of this thesis in mind.

1.1 Background

Sogeti (**So**ciété Pour la **G**estion de l'**E**ntreprise et **T**raitement de l'**I**nformation), an originally French IT company, founded in 1967 by Serge Kampf, is a part of the Capgemini group. Sogeti Netherlands is well known for its testing method TMap (Test Management approach) it has developed, this method has been apopted globally as a standard. Sogeti operates in a model where every expertise is divided in a *business line*, which currently totals in 16 different business lines. Figure 14 in Appendix B shows the distinct divisions and business lines of Sogeti Netherlands. This thesis was conducted at the Security business line. Sogeti's main services are the deployment of skilled employees in other companies as well as providing project based solutions with a fixed outcome. The Sogeti group currently has 20.000 employees, of which 2.500 are based in The Netherlands. Next to being deployed to a large number of client sites, Sogeti has a presence in 15 countries and over 100 locations worldwide.

2 Related Work

In recent years, there has been an increasing amount of literature on the topic of automotive security. Numerous studies focused on uncovering vulnerabilities in car systems, while others proposed solutions to prevent or detect these attacks. Some studies picked up where others left off, executing practical attacks on modern vehicles with the theoretical knowledge provided by other studies. In the following sections we will discuss some of the researchers' findings to give an idea of the current state of automotive security: Section 2.1 lists a number of theoretical and practical attacks executed by various research groups, Section 2.2 shows targeted attacks on the telematics units of various car manufacturers such as BMW, General Motors and Tesla, while Section 2.3 lists solutions discussed in various research papers. Finally, Section 2.4 provides an overview of some of the dependent and independent organisations currently working on the issue of security in automotive systems.

2.1 Practical attacks

There are a lot of shortcomings regarding the physical and wireless security of automobiles. Earlier research has shown that once an attacker gains *physical* access to a car, its security can easily be compromised [6].

In 2010, Koscher et al. [7] conducted an experimental security analysis of an unnamed modern automobile, uncovering and executing both physical and wireless attacks. The paper showed that, in contrast to what the researchers expected, the tested automotive systems were *tremendously fragile*. The typical car contains multiple communication channels like the CAN bus and groups different components together. For instance, powertrain components that generate real-time telemetry and other time-critical systems are interconnected in one CAN bus while another CAN bus controls less critical components like door locks and lights. From a security perspective it seems a good idea to physically separate these buses from each other, but in practice they are *bridged* to support subtle interaction requirements. Access escalation due to abusing this case of interconnected communication channels was explained and demonstrated in this paper. Furthermore, a simple fuzzing infrastructure which floods random packets onto the CAN bus, the standard communication channel for ECUs in a car, could easily execute a denial-of-service (DoS) attack leaving the CAN bus overloaded, thus useless. Since the CAN bus can be eavesdropped simply by connecting a reading device to the On-Board Diagnostics (OBD-II) port, executing a DoS attack is easy because the OBD-II also allows sending packets.

Moreover, the basic access controls implemented in the ECUs were frequently unused. For example, the firmware on an ECU controls all of its critical functionality and thus the investigated car's CAN protocol described methods for ECUs to protect against unauthorized firmware updates. The researchers demonstrated that they were able to load firmware onto some key

Vulnerability Class	Channel	Implemented Capability	Visible to user	Scale	Full Control
Direct physical	OBD-II Port	Plug attack hardware directly into car OBD-II port	Yes	Small	Yes
Indirect physical	CD	CD-based firmware update	Yes	Small	Yes
	CD	Special song (WMA)	Yes	Medium	Yes
	PassThru	WiFi or wired control connection to advertised PassThru devices	No	Small	Yes
	PassThru	WiFi or wired shell injection	No	Viral	Yes
Short-range wireless	Bluetooth	Buffer overflow with paired Android phone and Trojan app	No	Large	Yes
	Bluetooth	Sniff MAC address, brute force PIN, buffer overflow	No	Small	Yes
Long-range wireless	Cellular	Call car. authentication exploit, buffer overflow (using laptop)	No	Large	Yes
	Cellular	Call car. authentication exploit, buffer overflow (using iPod with exploit audio file, carphones, and a telephone)	No	Large	Yes

Table 1: Attack surface capabilities [8]. The mentioned *PassThru* channel uses tools like Ford's Vehicle Communication Module (VCM) which is a diagnostic device that works together with diagnostic tools for Windows to connect to cars through the OBD-II port [9].

ECUs, like the telematics unit - a critical ECU - and the Remote Control Door Lock Receiver (RCDLR), without any form of authentication. Similarly, the diagnostic protocol used by the OBD-II port should also make an attempt to restrict access to certain `DeviceControl` diagnostic capabilities. The research group was therefore also surprised to find that critical ECUs in the tested car would respond to `DeviceControl` packets without authentication first.

Finally, the researchers found that, in addition to being able to load custom code onto an ECU via the CAN network, it was very straightforward to design this code in a way to completely erase any evidence of itself after executing an attack. Thus, without such a forensic trail, it may be impossible to determine if a particular crash is caused by an attack or not. This was deemed to be a very dangerous capability too, as it minimizes the possibility of any law enforcement action that might deter individuals from using such attacks. Summarized, this experimental analysis showed so many design and implementation flaws it concluded that security of cars is virtually non-existent.

The follow-up research paper by Checkoway et al. progressed deeper into the unnamed car's systems and the comprehensive security analysis explained extended wireless attack vectors next to physical attack vectors [8]. Figure 1 shows the different channels the researchers used to gain complete control over a car. The figure illustrates the capabilities of a well-organized and well-funded group with decent knowledge of car systems in general and includes short- and long-range

wireless attacks next to the physical attacks introduced in the earlier paper. The researchers used the attack on the OBD-II port to be able to bridge into every communication channel in the car and to gain access to all ECUs. Next, the researchers extracted the firmware of all the ECUs of the car (around 30 ECUs for the researched model) and disassembled the firmware. Static analysis of the firmware code was performed and several vulnerabilities were found.

Another physical interface of the car, the entertainment system, was analyzed by the researchers. Where every modern car provides a user with a CD player able to interpret a mixture of audio formats, modern cars generally also provide the user with a USB interface to allow users to control their car's media system with their own device. This opens up new attack vectors for an adversary as it possibly allows to attack the media system of the car, for example by social engineering the target to play a malformed song or connect his personal media device to the car. Where taking over the CD player of a car alone might not do any harm, the researchers found that thanks to the interconnectivity of all systems that the CD player is a good attack vector for eventually attacking other automotive components through this interface. The researchers were able to create a WMA audio file such that, when burned onto a CD, it plays fine on a PC but sends random CAN packets when played by the CD player in the car.

Next to the vulnerabilities found through the physical interfaces of the car, the short-range wireless interfaces like RFID car keys, WiFi and Bluetooth posed to be attack vectors easily exploited by the researchers. The Bluetooth interface allows users' cellphones to connect to the car, for instance to call hands-free. Through reverse-engineering the researchers gained access to the operating system of the telematics ECU. They found that the telematics ECU uses a vulnerable implementation of the Bluetooth protocol stack. More specifically, this implementation made several calls to a `strcpy` function which is deemed unsafe due to its vulnerability regarding buffer overflows. The researchers wrote an exploit, gaining a shell on the telematics unit and consequently downloading more complex code through the 3G-connection of the car.

Finally, the long-range wireless channels also provided targets for the researchers to exploit. The telematics unit of the tested car has an implemented cell phone interface, supporting voice, SMS and 3G data. The critical telematics functions, like automatically dialing the emergency number when a crash is detected, are done over voice dialing as it provides connectivity over the widest area, whereas 3G coverage is not available everywhere. The researchers demonstrated an attack where a flaw in the aqLink software [1] of the telematics unit was exploited to authenticate a custom-made modem to the telematics unit, let it download an additional payload and execute it to gain control over the car remotely.

[1] Airbiquity's aqLink software allows data to be sent over voice channels instead of the common digital data channels: http://www.airbiquity.com/news/press-releases/airbiquity-releases-aqlink-5-band-modem/

2.1.1 Ford Escape and Toyota Prius

In an extended paper by Miller et al [10] the security of two modern popular vehicles, particularly the Ford Escape and Toyota Prius, were thoroughly investigated. Pointing at the previous research done, the researchers expanded on the ideas of what an attacker could do to influence the behaviour of a vehicle after a succesful attack. These attacks would allow arbitrary remote code execution on the ECU through one of the compromised channels mentioned in the earlier sections.

In particular, the researchers demonstrated how they were able to control some of the steering, braking, acceleration and display functions on the two tested vehicles. The researchers went into great detail explaining the attacks, the relevant messages over the CAN bus and all technical information needed to reproduce and understand the issues involved including source code and a description of necessary hardware to execute these attacks.

Although the researchers gained full access to all ECUs easily, understanding, observing, and reverse engineering the code and the messages sent over the CAN buses was far from easy. For example, in the Ford Escape, a certain CAN packet was observed including a byte to indicate how much the accelerator was depressed. However, this packet was being sent from the PCM (which reads the accelerator sensor) to the ABS, presumably to help it figure out if there was a traction control event in progress. It did not have anything to do with whether the car should speed up or not. There were countless examples like this including, for example, packets that indicate how much the brake is depressed but when replayed would not engage the brake. Understanding the actual meaning of all messages would take more research. Giving the assumption that a modern car runs on millions of lines of code [1], this was expected to be the case.

Another problem is that the receiving ECUs might have some safety features built into it that makes it ignore the packets the researchers were sending. For example, on the Toyota Prius, the packets used for turning the wheel in Intelligent Park Assist would only work if the car is in reverse. Likewise, packets for the Lane Keep Assist feature are ignored if they tell the steering wheel to turn more than 5%. It may be possible to circumvent these restrictions by tricking the ECU, but some extra work would be required, making these attacks a bit harder to execute.

2.2 Flaws in Telematics Control Units

Next to flaws in the physical channels of an automotive system, the different types of Telematics Control Unit (TCU) used in cars show enough flaws to exploit one of the vulnerabilities associated with these devices.

2.2.1 Chrysler and Uconnect

Very recently, Chrysler recalled 1.4 million vehicles after two security researchers revealed a software bug [11]. The paper by Miller et al., the same researchers who investigated the Ford Escape and the Toyota Prius mentioned in an earlier section, demonstrated several attacks that compromized the safety of a Jeep Cherokee thus the safety of a driver, using remote exploits [12]. This time the researchers used the Jeep Cherokee 2014 as a target. Figure 2 illustrates the archirectural diagram of the car. It stands out that also in this car, the radio system of the car including phone, Bluetooth and WiFi channels is interconnected to both CAN buses. Compromizing the radio would mean access to all the ECUs that control the physical attributes of the vehicle.

The 2014 Jeep Cherokee uses the Uconnect radio manufactured by Harman Kardon as the source for infotainment, Wi-Fi, GPS, apps, and cellular communication[2]. This system is widely available for different types of vehicles and includes a system on a chip (SoC) to provide its functionality. The Uconnect system also contains a chip to communicate with the two CAN buses: The CAN-IHS (Controller Area Network - Interior High Speed) and CAN-C (Controller Area Network - Control) buses. The researchers found several flaws in this Uconnect system. When looking at the WiFi interface, which is used to let users connect to a WiFi hotspot in the car, they they were able to reverse engineer the `WiFi.E:generateRandomAsciiKey()` algorithm that generates a password for the WPA2 protected WiFi communication channel. This algorithm uses time as an input for password generation and by estimating the time the amount of passwords to test was narrowed down to around 15 million. However, if no valid time was fetched before starting the WPA2 password generation algorithm, the date 1 January 2013 00:00 was used as an input. From there, researchers were able to narrow down the password to only a few options, making it trivial to find the correct WPA2 password.

This hack worked only when in close proximity of a car, so the researchers looked to find a way into the cellular communication channel. The ability to interconnect any device using the cellular network of Sprint [3] made it possible to connect a laptop tethered to the Sprint network to connect to an open port available on the Jeep. The researchers also found the range of IP addresses used in the Sprint network by the Jeep, and with a basic scan they found 2700 devices. They eventually opened a shell to the operating system of the TCU, and could send commands to the CAN bus remotely. By sending some commands the researchers were able to kill the brakes and the engine remotely. A few days after the paper was published, the Sprint network blocked the communication on the vulnerable port and Chrysler patched the issue. After that, Chrysler voluntarily recalled 1.4 million vehicles to patch the vulnerabilities.

[2] http://www.driveuconnect.com/features/entertainment/

[3] http://newsroom.sprint.com/news-releases/sprint-velocity-offers-automakers-customizable-approach-to-enhancing-new-and
htm

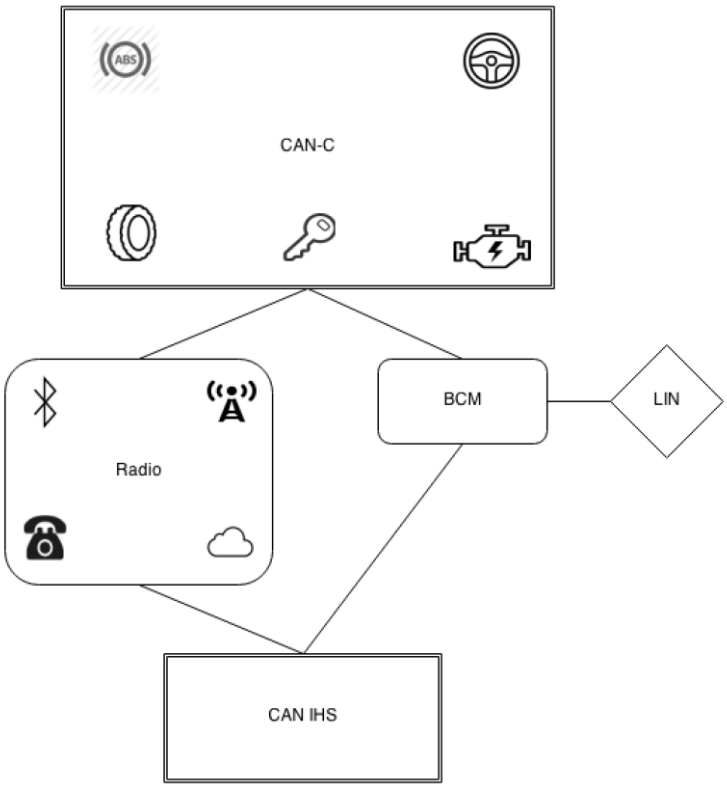

Figure 2: Jeep Cherokee 2014 architecture diagram [12].

2.2.2 General Motors' OnStar

Just after the Chrysler *hack*, a researcher exposed a vulnerability in General Motors' (GM) vehicles equipped with its telematics system OnStar [13]. After eavesdropping on the communication between the mobile application RemoteLink - which lets a driver control some features of their car, like locking doors and turning on lights - and the car, the researcher built a device called OwnStar that captured the communication between the application and the car. More specifically, the researcher jammed the frequency range of the car receiving the *open door* instruction but saving the information as well. With the car unable to read the signal due to the jamming, the user would again try to open the car with the application. The OwnStar device would then save the second signal and replay the first signal to the car so the user would get into the car. With this information of the second signal, the researcher was able to act as if he owned the car, finding its the exact location, unlocking the doors and even starting the engine [14].

While Chrysler voluntarily recalled 1.4 million vehicles after only a few days and patched a fix before the research was even published, General Motors had received a similar research paper in 2010, stating that the telematics OnStar used by GM was vulnerable to attack. GM took five years to actually address this issue and create a patch. The difference between these two cases

was the fact that the researchers did not publicly name the brand and mark of car they investigated. The unnamed car mentioned in [7,8], the two experimental analysis that were mentioned earlier in this section, was GM's 2009 Chevrolet Impala. The result is that GM took nearly five years to protect its vehicles from this hack, leaving millions of GM cars vulnerable to that attack that was privately announced to GM. This attack was a remote exploit that targeted the OnStar dashboard computer and was capable of everything from tracking vehicles to engaging their brakes at high speed to disabling brakes altogether [15].

Note that in this case, the weakness was not in the vehicle or TCU itself but in the application provided. The application communicated through SSL with the servers from GM, but the certificate of the server was not checked for validity. Spoofing the GM server and intercepting the communication uncovered that apart from the SSL connection, the data itself was not encrypted. In this way, the researcher used the information credentials for further access to the application and from there on, the car was compromised.

2.2.3 Tesla's Model S

Tesla's Model S is a state-of-the-art electric vehicle with lots of electronic components, including a 17" touch screen display which lets the user control a variety of functions of the car like control media, access navigation, enable the autopilot or adjust the height of the vehicle [4]. Users have to register for a Tesla account on the website of Tesla. With this account, for instance users can log in to the Tesla iPhone application to control functions of their vehicle with a mobile phone. Figure 3 shows a screenshot of the Tesla iPhone application. Locking and unlocking the car, locating it and controlling the roof are features of this application are amongst the functions of the iPhone application.

An article by Dhanjani in 2014 [16, 17] showed that the Tesla account can pose potential security issues due to a few aspects. First of all, the requirements for the password for a user account were six characters with at least one number and one letter. Combined with the fact that authentication by the user is *single-factor* and the website of Tesla did not seem to have any account lockout policy after a certain amount of incorrect login attempts, this requirement was vulnerable to a brute-force attack. After the article was published, Tesla bumped up the minimum password length to eight characters. Secondly, the Tesla REST API alows third party applications to use the credentials entered by the user to invoke the REST API on behalf of the user. This leads to the risk of malicious third-party applications being able to collect and abuse credentials of Tesla accounts. Until Tesla releases a SDK for third-party applications it was advised to Tesla owners not to use these applications. Finally, the WiFi connection used by the Model S showed services with open ports, like SSH, HTTP, NFS and telnet. However the researcher did not follow up on finding out more about these services, these services could

[4]Tesla Model S: http://www.teslamotors.com/models

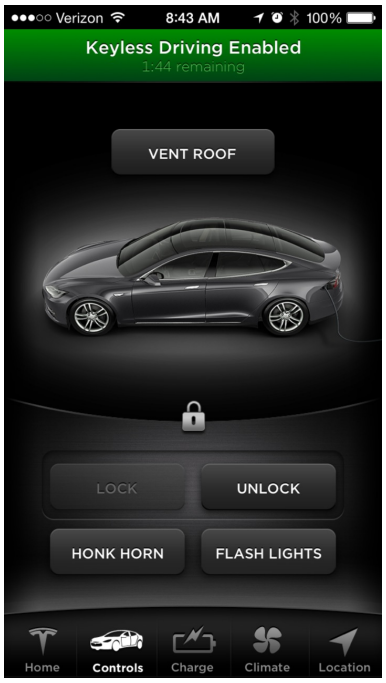

Figure 3: A screenshot of the Tesla iPhone application.

potentially be vulnerable to attack.

2.2.4 BMW's ConnectedDrive

BMW's ConnectedDrive technology offers features like remote services, personal assistance and real time traffic information [5]. The technology is embedded in a control unit, also called a Combox, and has been included in various BMW models since 2010. The Combox is responsible for playing music files from a USB stick, pairing with a Bluetooth device and includes a modem for GSM/GPRS/EDGE capabilities.

A researcher, asked by the German motorist's club ADAC to look at the privacy aspects of this device, published an article about the security vulnerabilities found in this control unit [18]. At first this device appeared to be well-secure, but after thorough investigation flaws were found. First of all, the emergency services offered by the device were encrypted using algorithms that are considered broken, like DES for example. Furthermore, after dumping and inspecting the firmware of the modem in the device, the researcher found that BMW used the same symmetric cryptography keys to communicate with the back-end server for *all* cars and that the private keys were easily extracted from the firmware. Also, some services of BMW's ConnectedDrive do not encrypt messages in transit between the car and the back-end server. Finally, the messages

[5] BMW ConnectedDrive: http://www.bmw.nl/nl/content/meer-bmw/bmw-connecteddrive/overzicht/connecteddrive-uitgelegd.html

sent are not salted so the messages are not protected against replay attacks.

According to BMW, the found vulnerabilities were confirmed and patched. Although users can not turn off ConnectedDrive technology themselves as it is integrated into the car system, a formal written request together with a visit to a garage is sufficient to disable it. Around 2.2 million cars worldwide were affected by these vulnerabilities [18].

2.2.5 Aftermarket Telematics Control Units

In a security analysis of a popular aftermarket Telematics Control Unit (TCU), Foster et al. found that besides security flaws in TCUs produced by the original equipment manufacturers, these TCUs also have vulnerabilities [19]. The automotive aftermarket is a big market in the USA and includes replacement parts and accessories for cars. The TCU is an accessory designed for add-on after the original sale of a car. This device looks like a dongle and is usually attached to a car's OBD-II port. It typically uses some form of low-power ARM core and incorporates accelerometers, a GPS chip, a cellular and/or Bluetooth modem, and a CAN transceiver chip.

The paper focused on analyzing one specific aftermarket TCU. The researchers were able to demonstrate both local and remote vulnerabilities and exploits using these vulnerabilities. In particular, the vulnerability of the TCU's SMS-based remote update procedure eventually provided the researchers arbitrary shell access, enabling them to compromize the entire car system. The bad design of the update protocol is illustrated in Figure 4. After examining logs created from initiating an update via SMS, the researchers could determine the entire procedure of the update:

1. The SMS command `rupd,USER,HOST,PORT,DIR` is sent to the TCU, which responds with `update,started`.
2. The TCU uses SCP to remote into the HOST on port PORT as user USER and retrieves `UpdateFile.txt` from DIR.
3. `UpdateFile.txt` is examined and files which have incorrect hashes or do not exist on the local system are then retrieved via SCP from the update server to a temporary directory on the TCU.
4. If the hashes of the new files match those found in `UpdateFile.txt`, the new files are moved to their target directories, otherwise the update process is restarted.
5. If `UpdateFile.txt` contains any of the following console commands, they are executed performing the appropriate action: `clear, identify, status, reset`.

The system uses the Secure Copy Protocol (SCP) to copy the files from the update server to the TCU. The researchers found quite a few weaknesses in the update procedure. First of

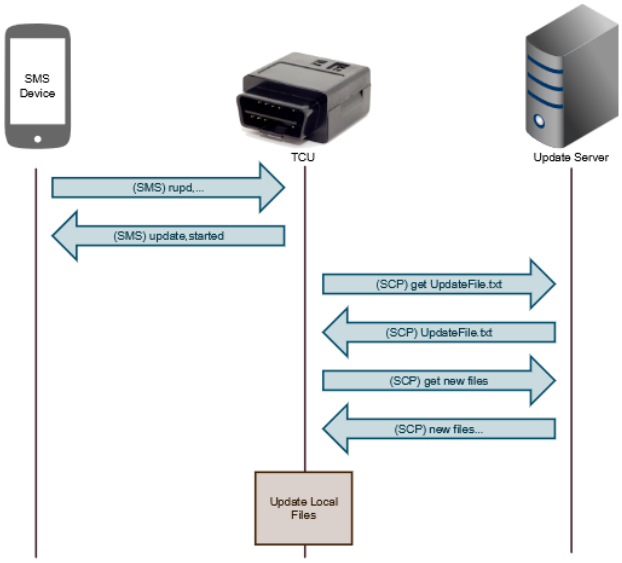

Figure 4: The flawed remote update procedure of a TCU. Source: [19].

all, the updates are not cryptographically signed in any way so it is easy to change the integrity of the update. Secondly, the TCU does not authenticate the server whereas the server does authenticate the device. Unfortunately, like in previous examples, all devices seem to share the same public/private keypair used for the update process. Finally, the update mechanism allows an arbitrary update server to be chosen, opening up more possibilities for the researchers to exploit. The researchers created a rogue update server that serves an update which immediately spawns a reverse shell and SSH tunnel to the victim TCU. The attack is illustrated in Figure 5 and goes as follows:

1. A remote update is initiated using the rogue server via telnet, web, or SMS.

2. The TCU downloads `UpdateFile.txt` containing `console.bak` (the original console binary), `console` (a shell script created which contains the attack), and the command `clear`.

3. After the TCU downloads all the files and replaces the system console command with the console script it calls console clear to clear the logs.

4. The console script starts and replaces itself with the original `console.bak`, starts a reverse shell and SSH tunnel, sends a SMS to the researchers informing that the attack was successful, and then calls the original console with the clear command.

5. Once the researchers receive the SMS from our script, or get a notification from the update server that the reverse shell is ready, they can SSH or tunnel into the device to get a root shell and access to the telnet and web interfaces.

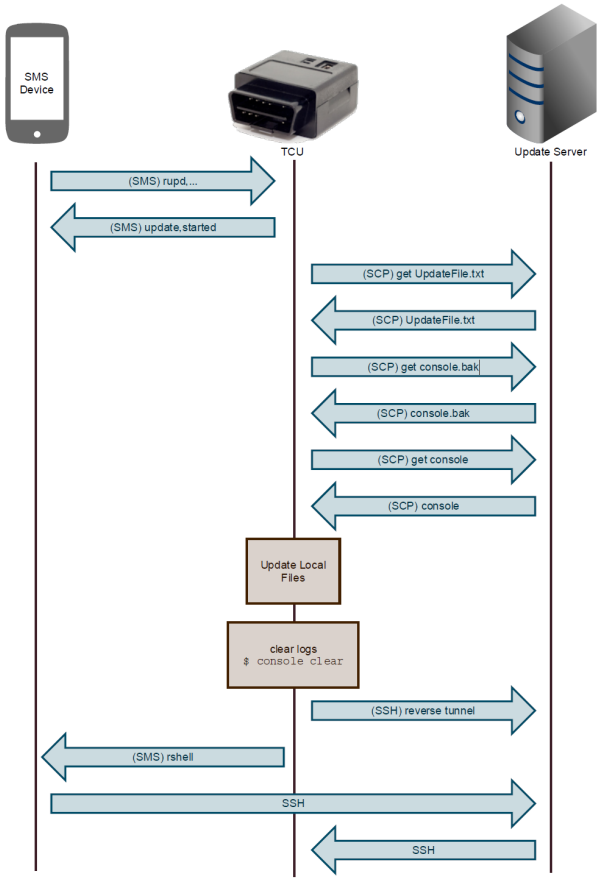

Figure 5: Remote exploitation via a malicious update [19].

To exploit the remote vulnerabilities on a larger scale the researchers needed to determine the addresses of the TCUs; either its IP address or the phone number associated with the SIM card. Some providers have Network Address Translation (NAT) to hide their client's IP addresses from the outside world. When this is not the case, the provided built-in web server of the TCU is accessible from the Internet and can be indexed by search engines. The researchers found that an Internet (Shodan) search for a particular string of words revealed several likely TCUs. Since all the TCUs from the same manufacturer use the same SSH server key, the server fingerprint search returned a considerable amount of devices. To scan for the associated phone number of the TCU, the researchers used the given fact that these SIM cards usually have a range of associated telephone numbers. Moreover, sending an SMS registration command to a suspected range of telephone numbers such as `status` would confirm a TCUs identity, making the creation of a a *war dialer* for enumerating these devices relatively straightforward.

2.3 Proposed Solutions

The uncovered security vulnerabilities listed in the previous subsections are just a small selection of vulnerabilities in automotive systems. Next to criticism, there was also room for constructivism. Consequently, most researchers did not only come up with vulnerabilities in the published articles but provided possible solutions to address, detect or prevent these very same vulnerabilities. The solutions range from basic, for instance two-factor authentication or not using outdated encryption methods, to more advanced solutions like using a protected REST API or using mobile NAT to hide public services from IP addresses of cars. However most proposed solutions are aimed at specific parts of the automotive system, none of them really propose details on how things should be fixed. From a car manufacturer's point of view, a forced recall is a method of last resort to address vulnerabilities and update firmware of a car, because it damages the reputation. (quote needed) (informal)

On a more specific note, there currently is little research available on how wireless communication should be secured, how it should be handled by internal systems and for instance, how OTA updates should be distributed safely and securely. Most research about this topic is high-level and does not describe the security-relevant details needed to successfully implement a secure system. Recent research showed different frameworks and mechanisms for over-the-air updates. In 2008, Nilsson et al. designed a way of securely updating firmware over the air called *secure firmware updates over the air* or SFOTA [20]. Though this protocol suits a great deal of the distribution of firmware updates, its assumptions include that the portal distributing the updates is well-protected and not considered a target for intrusion or denial-of-service attacks. Furthermore, the research does not cover sufficiently strong cryptographic primitives and does only include the over-the-air part, not the internal part of the securing the vehicle.

Later in 2008, Nilsson et al. [21] proposed a framework for self-verification of firmware updates over the air in vehicle ECUs. This framework provides verification of the firmware on an ECU after flashing it. It provides a method to battle *the time of check to time of use* attack explained in Section 4.7. It uses a hardware visualization technique called SPUMONE [22]. This framework has potential in cases where the target ECU has enough power, frequency and memory available -the target CPU used in the paper is the SH7780 with a SH-4A architecture- but most of the ECUs in a modern car are much more resource constrained, so this solution is not viable for our case.

In 2011, Idrees et al. provided a protocol for over-the-air firmware updates [23]. In this paper, a dedicated hardware security module (HSM), which can be a smartcard for instance, was introduced to handle the cryptographic requirements during an update. However, there it seems to be unpractical to equip all ECUs with a dedicated HSM. Next to unpractical, this solution would also be costly to implement as an increasing amount of ECUs are embedded into cars.

The protocol uses ideas and recommendations from two automotive security initiatives: EVITA and HIS, which will be elaborated later in this section.

In 2013, Studnia et al. analyzed multiple of the common security flaws and presented the security solutions currently being devised to address these problems [24]. Together with addressing the lack of suitable and strong enough cryptographic protocols for encryption, the paper focused on anomaly detection in the most used physical communication channel in a car: the CAN bus. The proposed system is simple but uses a lot of available resources. On every CAN bus, each frame identifier is associated to only one ECU. In other words, these frames can only be sent by one particular ECU and therefore cannot legitimately be sent by the others. Then, whenever a message is broadcasted on the bus, each ECU checks if the frame identifier is one of its own. If it is the case and if the ECU itself is not the actual sender of the frame currently being broadcasted, it immediately emits a high-priority alert frame to override the illicit broadcast.

In summary, various researchers proposed theoretical and practical solutions to prevent these basic attacks from happening. These preventive and detective measures should be known to any security-minded individual and it is again frightening to see the lack of basic security measures implemented in car systems by either Original Equipment Manufacturers (OEMs) or aftermarket companies, like including randomness into their security seystems or not using the same keypair for every car.

2.4 Initiatives towards better car security

Next to proposed solutions by individuals or research groups, there are some organizations that are or have been committed to address global security issues with automotive systems and look for means to improve the security. In this subsection, some of these initiatives are presented.

2.4.1 Automotive Open System Architecture (AUTOSAR)

AUTOSAR (AUTomotive Open System ARchitecture) is a worldwide development partnership of vehicle manufacturers, suppliers and other companies from the electronics, semiconductor and software industry. Since 2003, AUTOSAR has paved the way for innovative electronic systems that improve performance, safety and environmental friendliness. AUTOSAR has been pushing for standards in the industry and is aiming to be prepared for the upcoming technologies and to improve cost-efficiency without making any compromise with respect to quality. It also facilitates the exchange and update of software and hardware over the life of a vehicle. Next to the BMW Group, Bosch GmbH, Ford, General Motors, Toyota and the Volkswagen Group, more than 180 partners play an important role in the partnership. Only companies which have joined the AUTOSAR Development Partnership can use the designed AUTOSAR specifications free of charge.

To date, AUTOSAR has released multiple specifications and recommendations regarding performance, safety and environmental friendliness of automotive systems, though none of these relate to securing firmware updates. Besides, the AUTOSAR stack implementation requires a lot of extra resources that causes issues with more resource constrained ECUs. Since the automotive industry is highly cost-driven, companies usually settle for less quality. Futhermore, with the specifications being open to interpretation, most partners have their own proprietary implementations of these specifications, the opposite of the intention of this partnership.

2.4.2 E-Safety Vehicle Intrusion Protected Applications (EVITA)

The E-Safety Vehicle Intrusion Protected Applications or EVITA project had the objective to design, verify and prototype an architecture for automotive on-board networks where the security-relevant components are protected against tampering and sensitive data is protected against compromise when transferred inside a vehicle. The project was run by a partnership of companies such as Robert Bosch GmbH, BMW Research and Infineon [25]. The project ran from July 2008 until December 2011.

When the team started the project, several security requirements for automotive on-board networks were specified by looking at several defined use cases for automotive networks. After that, the goal was to design, verify and prototype an architecture for the on-board automotive network. The security functions of EVITA are split between software and hardware and the root of trust is placed in hardware security modules (HSMs) instead of on ECUs itself. For prototyping, Field-programmable gate arrays (FPGAs) were used to extend the standard ECUs with the functionality of cryptographic coprocessors. In order to ensure that the identified requirements are satisfied, selected parts of the secure on-board architecture and the communications protocols have been modelled using UML and verified using model-based verification tools. The resulting architecture and protocols were published as open specifications [26, 27]. Some of the specifications rely on custom-made AUTOSAR specifications.

As shown in Figure 6, there are three classes of EVITA: Full, Medium and Light. EVITA Full provides security for very critical applications requiring powerful security, for instance the headunit handling the communication from the car to the outside world. EVITA Full includes support for cryptographic primitives such as AES-128, Whirlpool, and ECC-256. For less critical automotive applications like communication between ECUs, EVITA Medium can be used. The Medium HSM is identical to EVITA Full except for stripping the ECC-256 and the WHIRLPOOL cryptographic primitives to suit less powerful ECUs. EVITA Light is more typical for sensors and actuators and only supports AES-128 symmetric encryption. The necessary shared secret can be established in various manners, for instance, by pre-configuration during manufacturing, by self-initialization or by running a key establishment protocol in software at the attached application processor.

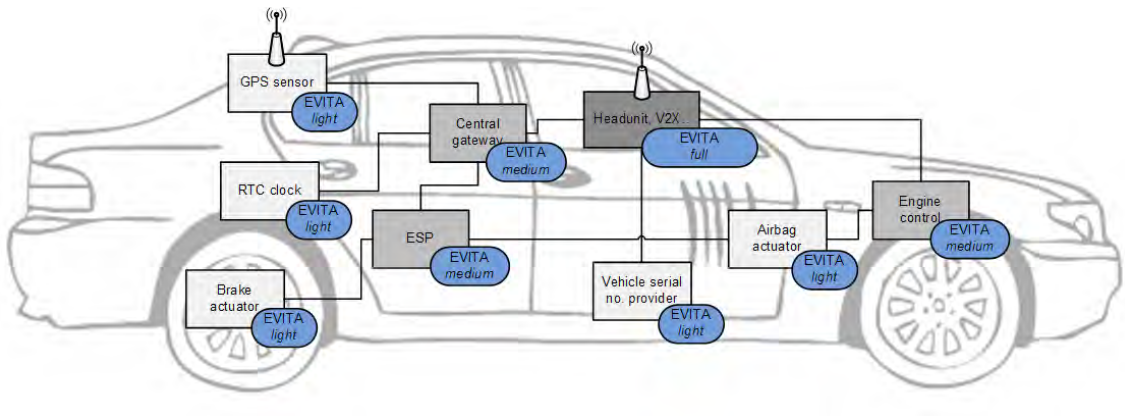

Figure 6: EVITA scheme: Full, Medium and Light [26].

2.4.3 Herstellerinitiative Software (HIS)

Herstellerinitiative Software (HIS) which translates to 'OEM software initiative' from German, is a consortium consisting of the car manufacturers Audi, BMW, Daimler AG, Porsche and Volkswagen. The initiative aims to standardize various aspects of software in ECUs. Among the works performed by the different groups in the consortium are software modules, software tests, software tools and bootloaders. With regard to flash programming, work has been done on reducing the time to flash a device and increasing the robustness of the flashing process [28]. The specification describes the protocols required to create a firmware flasher. Although HIS does not focus on security specifically, ideas and specifications of HIS are usable in this thesis.

3 Attacker model

Using the various examples mentioned in the earlier sections we can create our attacker model. In our attacker model, the attacker has physical and wireless access to the car. So the attacker can eavesdrop on internal communication, intercept communication, replay, change or inject messages into the untrusted channel to try and recover the communication. This model is an adaptation of the Dolev-Yao attacker model [29]. With these possibilities, an attacker could compromise the security of a car. It is important to realize that there is a difference between safety issues and security issues with a car. Where safety issues happen *at random*, computer security incidents happen due to malicious behavior. This makes car security different from car safety. Compromizing the security of a car usually leads to compromizing the safety of the car, whereas compromizing the safety of a car does not have to mean the security of a car is compromized. We will now take a look at the various adversaries that would benefit from obtaining, modifying or disrupting the firmware inside a car in any way.

3.1 Car Owners

Car owners are the biggest group of adversaries interested in modifying firmware. Despite being the biggest group, they are limited in both knowledge and resources. The majority of car owners do not possess the knowledge to mount an advance attack on a car system for own benefit. Then again, usually people are only interested in increasing the power output of their engine and they will look at possibilities to get extra content or performance for their vehicle without having to spend too much time or resources on it.

It is not very unrealistic that in the future, several new car features can be *unlocked* by paying extra for it. With current (mobile) payment options it is fairly straightforward to buy additional features for cars. Newer car models already offer extra features on a subscription base, like voice dialing, navigation, and remote diagnostics. Examples of these subscription-based services are General Motor's OnStar, Chevrolet's MyLink and BMW's Assist. Improved steering, better engine performance and sophisticated braking could be features that would be accessed by payment or a subscription in the future. A firmware update to several ECUs could unlock these features, for instance.

We have seen several methods of identification of a user to a backend portal. Some examples of identification and authentication are the Vehicle Identification Number (VIN), a user account, the mobile phone number associated with the car, a public/private keypair or a combination of any of these methods. When a car owner wants extra content or features for his car without wanting to pay for it, he will possibly try to circumvent measures of authentication or identification. For instance, sharing the same user account with additional privileges in multiple cars.

3.2 Engine Tuners

Car tuning is modification of the performance or appearance of a vehicle. Where most vehicles leave the factory set up for an average driver's expectations and conditions, car tuning has become a way to personalize the characteristics of a vehicle to the owner's preference. Cars may be altered to provide better fuel economy, produce more power, or to provide better handling. Since ECUs are embedded into cars, engine tuning exists.

There are two ways to increase the performance of a vehicle by (ab)using ECUs. One is chip tuning, where performance chips are placed between engines and actuators, altering the data that ECUs receive and hereby changing the performance of the engine. ECU remapping is an adjustment of the firmware of the engine, either by writing own firmware or changing the firmware of the manufacturer. Both options are performed to yield optimal performance, to increase the engine's power output, economy, or durability. These goals may be mutually exclusive, and increased performance may decrease the engine life due to more stress on engine components. Since the introduction of the obligatory OBD-II port in 1986, ECU remapping has become more popular in contrast to chip tuning. Since ECUs have been embedded into cars, engine tuning has turned out to be a benefitial industry.

ECU remapping is available for almost all modern cars. Usually a chip is desoldered and the firmware is extracted, disassembled and analyzed. After that, parameters of the firmware are tested to optimize the performance of a car. Next, the new firmware is flashed to the ECU through the OBD-II port. Once a performance or tuning pack is available for a car type, it can be used for all cars of the same type. Being the first to create a flash pack for a new type of car can earn a lot of prestige next to income, so the incentive of car tuners is more than just money.

Being able to tune modern ECUs is becoming increasingly challenging due to the waging *arms race* between the car manufacturers and engine tuners [30]. Attacks are becoming more advanced due to so-called anti-tuning protection included in modern ECUs. Engine tuners need to have extensive knowledge of cars, the communication buses of cars and ECUs to successfully extract, modify and re-insert firmware. Groups of engine tuners therefore will have to invest a lot of time and resources to be able to create a tuning pack.

3.3 Security Researchers

Security researchers have other intentions when trying to obtain, modify or disrupt firmware. Security researchers or research groups are usually well-funded, resourceful and independent, i.e. have no bias towards any other entity. Their research is valuable to the affected community and companies often are disclosed useful information from security research about their products. For instance, the research group testing the *unnamed* sedan notified General Motors before

publishing their research, which showed a lot of flaws in the firmware used by GM in their cars. Security researchers are driven by their eagerness to learn new things and to test hypothesis. In the end, from a manufacturer's point of view, they offer a great service by uncovering potential vulnerabilities and exploits, usually for free. However due to several clashes with the disclosure of vulnerabilities, security researchers also sell their findings on the zero-day market [31].

3.4 Black Hat Hackers

A black hat hacker is a hacker who *violates computer security for little reason beyond maliciousness or for personal gain* [32]. Black hat hackers keep the awareness of the vulnerabilities to themselves and do not notify the general public or the manufacturer for patches to be applied. When driven by personal gain, black hat hackers would be interested in obtaining firmware for cars for reselling it to government agencies, terrorists or competing car companies, or for exploit creation or other malicious activities. Selling these vulnerabilities as zero-days can be quite lucrative [31]. Once black hat hackers have gained control over a system, they may apply patches or backdoors to car systems only to keep their reigning control.

3.5 Ethical Hackers

Ethical hackers act within the same principles of security researchers, but can apply more means to compromize a system, like social engineering. Ethical hackers can try and test security unannounced and disclose the found vulnerabilities using *responsible disclosure*, for instance. In contrast to black hat hackers, ethical hackers do not violate computer security for personal gain but to make the world a safer place. When acting on behalf of a contract, ethical hackers could have a lot of resources and time at their disposal. Their knowledge or eagerness to learn new things drives them to try and find vulnerabilities. In this case, ethical hackers can try to obtain car firmware to reverse engineer and test the firmware for vulnerabilities, or test if the firmware can be extracted in the first place.

3.6 Car Thieves

Car thieves are driven by personal gain and usually do not have any knowledge or resources available to steal a car, for instance. However, professional crime syndicates often have quite a lot of resources available. These syndicates will generally use exploits created or sold by black hat hackers. With these resources, they would be able to equip car thieves with devices to remotely unlock cars or disabling the alarm or GPS by plugging in a device to the open OBD-II port of a car. An example would be a car thief that works at a valet parking, targets expensive vehicles, parks them, modifies the firmware by plugging in a device to the OBD-II port so the car can be tracked and remotely unlocked at a later time.

3.7 Competing Car Companies

Car companies have entire research departments dedicated to optimizing their cars' behavior. The resources and knowledge at their disposal is sufficient to mount attacks like desoldering chips or man-in-the-middle attacks to obtain the firmware of a car of a competing company. This firmware can give the company insights on how aspects like fuel distribution, engine optimization and brake pedal sensing are handled by the firmware. Of course, legal issues prevent car companies from officially employing such *strategies*. Though, with the recent scandals in the automotive industry, it would not be surprising if these tactics were already employed unofficially.

3.8 Terrorists

Terrorists could be able to use exploits gained by black hat hackers or disclosed by researchers to create chaos or fear, for example mass disabling one type or brand of car in a city to cause a traffic breakdown or mass accidents. Terrorists can be state-sponsored actors, making the resources and know-how at their disposal virtually unlimited. This would enable terrorists to mount brute-force attacks against the wireless vehicle infrastructure for example, and hereby possibly causing various incidents.

3.9 Summary

Adversary	**Resources**	**Knowledge**	**Time**
Car owners	*	*	*
Engine tuners	**	***	**
Security researchers	**	***	**
Blackhats	*	***	**
Car thieves	*/***	*	*
Ethical hackers	*/**	***	**
Terrorists	***	*/***	***
Competing car companies	***	***	***

Table 2: Summary of possible adversaries

Table 2 shows the various adversaries and their available resources, knowledge and time to mount attacks to "get what they want". Competing car companies, terrorists and car thieves backed by crime syndicates usually have the most resources at their disposal. When it comes to knowledge, car owners, car thieves and terrorists work with what they are supplied with. Since terrorists and competing car companies have the most resources available they likely have more time to work on attacks as well.

4 Required Security Properties

With the available research, the recent car hacks and the adversaries willing to put resources into attacking the car to obtain, modify or disrupt the firmware of a car, we can now define the required security properties that need to be addressed to secure a firmware update. Next to the standard requirements regarding information security, the CIA triad (Confidentiality, Integrity, Availability), we need several other security properties to secure our update model.

4.1 Confidentiality

Our first required security property is confidentiality, which stands for preventing data from disclosure to unauthorized parties: keeping this data *confidential*. In information security, data encryption is a common method of ensuring confidentiality. Encryption is very widespread in today's environment and can be found in almost every major protocol in use. A very prominent example is SSL/TLS, the security protocol for communication over the Internet. This protocol is used to ensure confidentiality but also integrity, for instance. Another measure of keeping data confidential is the use of access control methods to restrict access to the data only to people that are authorized. Usually, sensitive data is categorized into classes. The more sensitive the data, the higher the class and the stronger the used encryption methods and authorization methods to keep the data confidential.

For our model we need confidentiality to prevent attackers from reading the data that is communicated over an insecure channel.

4.2 Integrity

In information security, data integrity stands for maintaining and assuring the accuracy and completeness of data over its entire life-cycle. This means that data should not be modified in an unauthorized or undetected way, as data that has been tampered with could be useless or even malicious. In addition to message confidentiality, information security systems typically provide message integrity. There are various ways of assuring integrity of a message, for instance during transport over an insecure or unstable communication channel. Often, a hash function is used to create a hash value for a certain block of data. The hash value over the data is calculated on both the sender and the receiver's side. If both resulting hash values match, the integrity can be guaranteed in a high probability, meaning that the chance the data was altered is negligible. However, a hash function does not *guarantee* integrity. Moreover, securely sending the hash value for a message over a untrusted channel is another challenge. Most recent hash functions provide pre-image resistance, second pre-image resistance and collision resistance so that it becomes infeasible to break the system. When transfering data from one system to another using a serial communication channel like the CAN bus, other means of assuring integrity to a certain extent

is provided, for instance by using a cyclic redundancy check (CRC). However, the CRC was originally designed to detect *accidental* changes in the data. To really provide message integrity, we need something stronger. A message authentication code (MAC) provides a message with integrity next to authentication.

For our model we need integrity to assure that the data that is transfered from the sender from the receiver is not modified (by an attacker) in any way.

4.3 Availability

For our model we need availability to allow authorized users to be able to access data when needed. Nowadays, denying access to data has become a very common attack. Usually, a distributed denial-of-service (DDoS) attack is the cause of data being unavailable. Such downtime can be very costly. Next to that, general DoS attacks are perceived to be very annoying. Other factors that could lead to unavailability of important data may include power outages or natural disasters such as floods and fires. Redundancy is a way to ensure availability, but only up to a certain level. Regularly doing off-site backups can limit the damage to (digital) infrastructure by natural disasters or power outages. For data services that are highly critical regarding uptime, multiple copies usually exist and are being kept synchronized. Sometimes, availability cannot be guaranteed. For instance, if a car is in an underground parking lot without reception to the mobile network, updates cannot be applied. On the CAN bus, the most used communication channel inside a modern car, availability is an issue as well. A device sending a certain packet with a *dominant* bit forces the bus in a dominant state. Consequently, every time another device wants to send, the bus arbitration algorithm tells it to back off, resulting in no message getting on the bus.

For our model we need availability to assure the updates can be transfered over the air and to assure the updates are applied from the telematics unit to the corresponding ECU.

4.4 Authenticity

Authenticity is the assurance that a message, transaction, or other exchange of information is from the source it claims to be. Authenticity involves the proof of identity and can be assured through authentication. The process of authentication usually involves more than one *proof* of identity. This proof might be something a user knows or possesses, like a password or a device like a keycard. Modern systems can also let a user provide proof based on something he is. Biometric authentication methods for instance, include aspects like fingerprint or retinal scans.

For user interaction with systems, programs, and each other, authentication is deemed critical. A user ID and a password input is the most used method of authentication. However, this method of authentication introduces a variety of problems as well. Passwords can be simply

brute-forced if the passwords are not long enough or not complex enough. Also, remembering dozens of passwords for as many applications can be frustrating for users. Furthermore, when a user ID and password combination is compromised, adversaries can claim to be someone they are not. Hence, two-factor or multi-factor authentication is more common for enterprise and critical applications and systems. Multi-factor authentication systems can use key cards, USB tokens, mobile devices or biometric data. For system to system authentication, Public Key Infrastructure (PKI) Authentication is a commonly used method. For instance, SSL connections to websites do not only provide encryption but also verification that the web site is authentically the site it claims to be.

For our model we need authenticity to prove that the systems we use are in fact the systems they claim to be. We need authentication both ways, so-called mutual authentication, to ensure that the correct sender is communicating with the correct receiver. It prevents so-called man-in-the-middle (MITM) attacks, where an attacker intercepts and/or relays communication between the sender and receiver and controls the entire conversation.

4.5 Forward Secrecy

Forward secrecy (FS; also known as perfect forward secrecy, or PFS) is a property of key-agreement protocols like Diffie-Hellman to ensure that a session key derived from a set of long-term keys cannot be compromised if one of the long-term keys is compromised in the future. The key used to protect transmission of data must not be used to derive any additional keys, and if the key used to protect transmission of data is derived from some other keying material, then that material must not be used to derive any more keys. In this way, the compromise of a single key permits access only to data protected by that single key. The Diffie-Hellman key exchange protocol can use *ephemeral* keys to ensure forward secrecy.

For our model we need forward secrecy to prevent an attacker from intercepting data from a used untrusted channel and later decrypt it using a compromised long-term key.

4.6 Private Key Protection

When using a public-key cryptosystem, a compromized private key can have disastrous effects. Obviously, all cryptosystems rely on this private key to remain private. We have seen some cases where all private keys on the client side were the same [30]. If such a key gets compromised, the entire system of encryption collapses like a house of cards. When storing a private key on a microprocessor, there is one main risk: an attacker attempting to compromise the ROM and extracting the key. This can be done in various ways. One way is an attacker attempting to compromise the ROM location where the keys are stored with a bit flip. Another way is attempting to extract the key from the ROM location without leaving a trace of device tampering. A

redundant copy of the private key can be placed in the ROM at manufacturing time to prevent this type of attack [33].

For our model we need private key protection because our entire system will depend on its privacy. If an attacker manages to obtain the private key of a TCU, ECU, or even the portal, he can obtain the firmware as well.

4.7 TOCTTOU attack protection

A TOCTTOU (time of check to time of use) is a race-condition attack that exploits the time between a race condition being checked and the result of this check being used to perform an action. An examplary attack is described in [34], where a UNIX *symlink* (symbolic link) is changed between validating this symlink and using it to read or write data, assuming the outcome of the check is still valid. Operating systems have schedulers that designate processes time to execute. While the first process does the check, a second process may interfere and use this very same symlink whereas the original process assumes that it is using the resource exclusively.

In the case of a firmware update, this timing attack can take place between checking the validity of the update on the microprocessor and flashing the update to the microprocessor, replacing the old version.

4.8 Randomness

In information security, randomness is a very important variable that needs to be included in encryption mechanics. Kerckhoff's principle mentions that we cannot rely on the secrecy of algorithms, only on the secrecy of keys [35]. The safety of keys also relies on the quality of random numbers. Without properly generated randomness or initialization vectors (IVs), every message with the same content will look the same. Not only to the sender or the receiver but also to an attacker eavesdropping on the communication. Two ciphertexts with the same message should be indistinguishable, and an adversary should not be able to derive information about a message when only having access to the ciphertext and public key. A pseudorandom number generator (PRNG) or true random number generator (TRNG) is used to generate sequences of numbers whose properties approximate the properties of sequences of random numbers [36]. If the output of a PRNG is even remotely predictable, it reduces the security of the entire encryption scheme.

In Section 2 we have seen that besides randomness in encryption, randomness in phone numbers of SIM cards in TCUs is useful, to mitigate scanning and wardialing attacks on IP ranges or phone number ranges. Furthermore, the randomization of private keys during manufacturing, chassis numbers (VIM numbers) for a range of vehicles should harden the attacker's ability to obtain any information at all if these keys or numbers are used for identification of a vehicle.

For our model we need randomness in every message that we send to prevent an attacker from obtaining information about the plaintext without knowing the private key. Such a plaintext attack, either a chosen-plaintext attack (CPA) or known-plaintext attack (KPA) is aimed at reducing the security of the encryption scheme.

4.9 Summary

Required property	**Prevents attack**	**Achieved by**
Confidentiality	Eavesdropping	Encryption
Integrity	Firmware tampering	Hashing, CRC
Availability	DoS attacks	Anomaly detection, backups, timeouts
Authenticity	MITM attacks, spoofing	Mutual Authentication, certificates
Private key protection	Key extraction	Key redundancy, ROM protection
Timing attack protection	TOCTTOU	Atomicity, constant time algorithms
Randomness	Plaintext attacks	PRNG, TRNG, ciphertext indistinguishability

Table 3: Summary of required security properties for our model

To summarize, our model needs to assure various aspects regarding information security. Table 3 shows the required properties to create a model that is robust against the attacks listed in the second column. Finally, the third column shows the possible means to achieve the corresponding security properties or prevent the attacks mentioned in the second column.

5 Target Platform and Cryptography

Our proposed solution uses an ECU that is constrained by size, power, speed and even temperature requirements [37]. Also, the required security properties discussed in the previous section need to be taken into account when selecting the most suitable primitives. Modern cryptography offers lots of variety in primitives to pick from. The choices made for various aspects of our model are explained in this Section.

5.1 CPU

The microprocessor for this thesis is of the MPC5500 family and uses a FreeScale MPC5566MVR132 32-bit microcontroller as a specific processor. It offers a single core processor with speeds up to 132 MHz, 128KB of SRAM, 32KB data cache and 3MB of flash memory. These microprocessors are used in automotive applications such as engine control and transmission control systems. For instance, Bosch GmbH uses this family of microprocessors in the EDC-16 series of fuel injection controllers. The MPC5566 uses the PowerPC architecture, more specifically a e200z6 core. This e200z6 has a branch prediction unit, a 32-entry MMU, signal processing extensions (SPE) and 32 KB L1 data cache. It can use the complete 32-bit PowerPC instruction set [38].

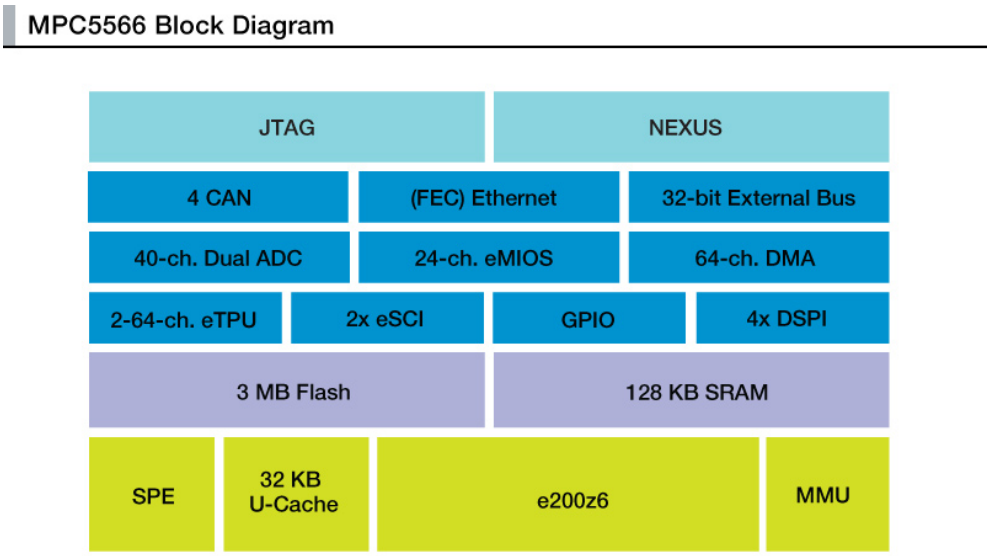

Figure 7: MPC5566 Block Diagram. The MPC includes 3MB flash memory and 128KB of SRAM.

5.2 Storage

Although the specific CPU has some memory available for storage, the storage is not nearly enough to run an operating system with hardware virtualization on top of it. There is enough

storage to put quite a large binary application in the memory, and there is even space for a backup. When flashing occurs, there are two versions of the firmware in the flash memory: the old version and the new version. For ECUs with smaller applications there even is space enough to do a *factory reset* when necessary. In this case, during a firmware upgrade, the amount of firmware versions on an ECU would be three: the factory version, the old version and the new version. When an integrity check fails on the old or new firmware, the firmware should be replaced with the factory firmware.

5.3 Delivery

There are various methods to communicate with a vehicle from the outside world, for instance through the mobile network (GSM/LTE), through a Bluetooth connection, or through a wireless network router. The available communication channels depend on the TCU that is included in the car. Inside the car, there are several communication systems available to deliver the firmware to the correct ECU. Apart from the CAN protocol, which is extremely vulnerable to denial-of-service attacks, the Local Interconnected Network (LIN), FlexRay, Media Oriented Systems Transport (MOST) and even Ethernet [39] are alternatives for transportation of the firmware. However, since the method of delivery is not a factor in our model, we refrain from any specifics regarding the delivery methods.

5.4 Retry Timeout

We propose the implementation of a timeout for a retry of communication when an integrity check or authenticity check fails. Also, when a key exchange fails, there will be a timeout before a device can retry to authenticate itself. Since there are legitimate cases where a check or key exchange fails, like a temporary glitch in the wireless communication due to limited coverage, the timeout period should increase as the number of failed attempts grows. To prevent brute-force attacks like the one described in [40] where a 2-byte authentication seed was used in combination with no brute-force protection, these timeouts should definitely be implemented.

5.5 Key sizes

NIST, the National Institute of Standards and Technology recommends the use of certain key sizes for various symmetric and asymmetric (public- and private keypair) algorithms. These recommendations are shown in Table 4. The first column of this table represents the minimum strength in bits that the corresponding algorithms in the other columns provide. 2TDEA is the 2-key triple-DES algorithm (TDEA stands for Triple Data Encryption Algorithm), and 3TDEA is the 3-key triple-DES algorithm. The third and fourth column show the the corresponding key sizes of asymmetric or public-key algorithms Rivest Shamir Adleman (RSA) and Elliptic Curve Cryptography (ECC) to achieve the minimum strength in bits in column one. NIST guidelines state that ECC keys should be twice the length of equivalent strength symmetric key algorithms.

Of course, these estimates assume that no major breakthroughs in solving the underlying mathematical problems that ECC or RSA are based on will be found. If we look solely at key size, symmetric cryptographic algorithms would be the right choice because they offer the best protection while using the smallest keys.

Minimum Strength (bits)	**Symmetric Algorithms**	**RSA (bits)**	**ECC (bits)**
80	2TDEA	1024	160
112	3TDEA	2048	224
128	AES-128	3072	256
192	AES-192	7680	384
256	AES-256	15360	512

Table 4: NIST-recommended key sizes of cryptographic algorithms [41]

5.6 Protocols

To add the various parts of information security to our model, we need to implement some cryptographic protocols to assure confidentiality, integrity, and authenticity.

5.6.1 Symmetric vs Asymmetric cryptography

There has been written much about the use of symmetric and asymmetric key ciphers. Secret communication using asymmetric encryption is possible even if the sender's and receiver's key are public, as long as the sender's and receiver's private keys are kept private. On the contrary, symmetric ciphers are less resource-consuming because asymmetric ciphers need more mathematical operations to offer the same security. Both approaches have its advantages and disadvantages. Often, a combination of symmetric and asymmetric cryptography is used for security solutions. For instance, two parties may agree on a shared secret resulting of a Diffie-Hellman key exchange. This shared key can be directly used as a key or used to derive another key which can then be used to encrypt further communications using a symmetric key cipher, for instance AES. In our model we will use this combination to provide confidentiality, forward secrecy, authenticity and integrity. More specifically, for asymmetric cryptography, we will use ECC, as it has a smaller keysize and outperforms RSA, especially on embedded devices [42].

5.6.2 SHA-2

Often, cryptographic hash functions are used to assure the integrity of messages by providing a checksum of the message. Hash functions are functions that map data of variable size —for instance a message— to data that has a fixed —and usually smaller— size. For example,

computing the hash value of a message and comparing the result with a published hash result can show if a message has been tampered with or not. There is a variety of hash functions available. Some hash functions are more secure than others, for instance MD5 and SHA-1 are considered to be broken [43, 44]. The Secure Hash Algorithm 2 (SHA-2) is a family of cryptographic hash functions and provides functions in hashes of 224, 256, 384 or 512 bits: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256. The SHA-2 hash function is implemented in security applications and protocols such TLS and SSL, PGP and SSH. Although there are some attacks known for some rounds of the hash function, there are no known attacks against full-round SHA-2. So the security of SHA-2 is deemed sufficient for now and is included in the National Institute of Standards and Technology (NIST) Federal Information Processing Standards Publication (FIPS) [45]. Several implementations of the algorithm have been validated by the Cryptographic Module Validation Program (CMVP) [46].

5.6.3 Elliptic curve Diffie-Hellman

Elliptic-curve Diffie-Hellman (ECDH) is an anonymous key agreement protocol that allows two parties, each having an elliptic-curve key pair, to establish a shared secret, for instance for Alice and Bob, over an untrusted channel. It is a variant of the Diffie-Hellman protocol using elliptic curve cryptography (ECC), which is an approach to public-key cryptography and uses *elliptic curves* over *finite fields*. The elliptic curve is a curve with the corresponding equation $y^2 = x^3 + ax + b$ along with a point at infinity ∞. Its domain parameters are (a, b, p, G, n, h), where a and b are the curve parameters modulo p. G is a point of order n and by definition $nG = \infty$. The integer h is the cofactor of n (the ratio between the number of points on the curve and the number n). The entire security of ECC depends on the ability to compute a point multiplication and the inability to compute the multiplicand given the original and product points, the discrete-logarithm problem or DLP. An attacker who can solve the elliptic-curve discrete-logarithm problem (ECDLP) can figure out Alice's secret key from Alice's public key, and can then compute the shared secret the same way Alice does. Alternatively, an attacker can figure out Bob's secret key from Bob's public key, and can then compute the shared secret the same way Bob does.

5.6.4 Curve25519

Curve25519 is a Diffie-Hellman function suitable for a wide variety of applications. Assume that we have Alice and Bob that want to use the Curve25519 function to communicate. Given Alice's 32-byte secret key `ska`, Curve25519 computes Alice's 32-byte public key `pka`. Given Bob's 32-byte secret key `skb`, Curve25519 computes Bob's 32-byte public key `pkb`. With the Alice's 32-byte secret key and Bob's 32-byte public key, Curve25519 computes a 32-byte secret `ss` for Alice. This secret key is shared by the two users, and can then be used to authenticate and encrypt messages between Alice and Bob. Curve25519 is constructed in a way that it avoids many potential implementation pitfalls. For instance, it avoids any problems with poor random

number generators [47]. This selected curve is also listed as being a safe curve on the SafeCurves page [48]. Curve25519 offers 128-bit security, which confirms the factor of two used earlier in Table 4.

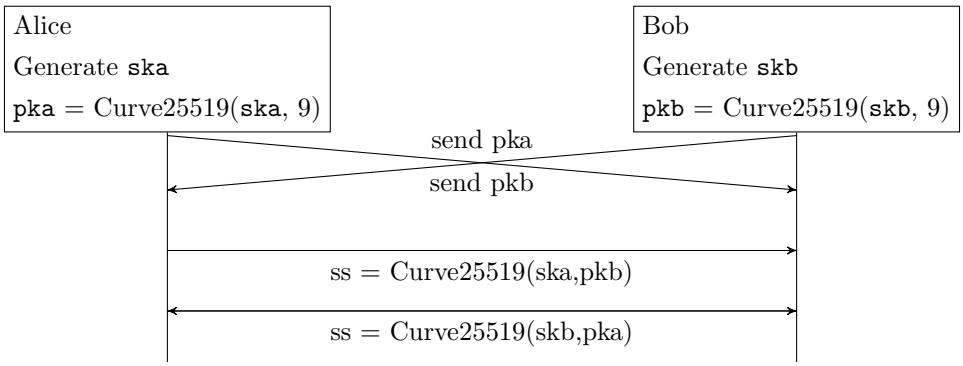

Figure 8: A view of the Elliptic-curve Diffie-Hellman (ECDH) Curve25519 function

Figure 8 shows the data flow between two parties from a point of each party possessing a secret key to both parties possessing the shared secret using the Curve25519 function.

5.6.5 Salsa20

Salsa20 is a stream cipher, which was designed in 2005 by Daniel J. Bernstein and submitted to eSTREAM [49]. As of September 2011, Salsa20 is included in the eSTREAM portfolio. The Salsa20 encryption function is a long chain of three simple operations on sixteen 32-bit words [50]:

- 32-bit Addition, producing the sum $a + b$ mod 232 of two 32-bit words a, b.
- 32-bit Exclusive-or, producing the xor $a \oplus b$ of two 32-bit words a, b.
- 32-bit Rotation, producing the rotation $a \ll b$ of a 32-bit word a by b bits to the left, where b is constant.

Salsa20 expands a 256-bit key, a 64-bit nonce and a 64-bit stream position into a 512-bit stream. Salsa20 uses 20 rounds on the 32-bit words to acquire this stream. The cipher encrypts a b-byte plaintext by xor'ing the plaintext with the first b bytes of the stream and discarding the rest of the stream. It decrypts a b-byte ciphertext by xor'ing the ciphertext with the first b bytes of the stream. There is no feedback from the plaintext or ciphertext into the stream. This gives Salsa20 the advantage to seek to any position in the output stream in constant time and that blocks can be computed in parallel. In comparison to AES —which is a block cipher but also includes modes to stream—, Salsa20 is fast in software implementations as there is little time needed for key setup and it has one of the lowest cycles-per-byte count for encryption [51]. Just like AES, Salsa20 can be used together with an authenticator for authenticated encryption. It is often used with the Poly1305 authenticator, which will be explained in the next subsection.

Next to the 20-round version of Salsa20, there are 8-round and 12-round variants of Salsa20 with even faster speeds available, though they offer less security.

5.6.6 Poly1305

Poly1305 computes a 16-byte authenticator of a variable-length message m, using a 32-byte one-time key. It produces a tag that authenticates the message such that an attacker has a negligible chance of producing a valid tag for a inauthentic message. The message m is broken into 16-byte chunks which become coefficients of a polynomial in r, evaluated modulo the prime number $2^{130} - 5$, which explains the origin of the name of the authenticator [52].

The 32-byte one-time key is partitioned into two 16-byte chunks and represented as a pair (r, s). The first 16 bytes of the one-time key form an integer, r, as follows: the first four bits of the bytes at indexes 3, 7, 11 and 15 are cleared, the bottom 2 bits of the bytes at indexes 4, 8 and 12 are cleared and the 16 bytes are taken as a little-endian value. An accumulator is set to zero and, for each chunk of 16 bytes from the input message, a byte with value 1 is appended and the 17 bytes are treated as a little-endian number. If the last chunk has less than 16 bytes then zero bytes are appended after the 1 until there are 17 bytes. The value is added to the accumulator and then the accumulator is multiplied by r, all mod $2^{130} - 5$. Finally, the last 16 bytes of the one-time key s, are treated as a little-endian number and added to the accumulator, mod 2^{128} to keep the result at 16 bytes. This result is serialised as a little-endian number, producing the 16-byte tag [53].

The 32-byte one-time key which is used as input, can be the result of any arbitrary keyed function, like AES or Salsa20. Its security also relies on the keyed function that is used for generation of the 32-byte one-time key. The pair (r, s) should be unique, and must be unpredictable for each invocation of the function. The s should be unpredictable, but it is perfectly acceptable to generate both r and s uniquely each time. Because each of them is 128 bits, using a PRNG to generate hem is also acceptable. Recently, Poly1305 has been selected together with the ChaCha20 symmetric cipher (a variant of Salsa20) for use with TLS/SSL and this combination has been added to OpenSSH too [54]. Poly1305 can be computed at high speed in various CPUs and has optimized implementations for Athlon, Pentium, PowerPC, and UltraSPARC processors [55].

5.6.7 Ed25519

Ed25519 is a specific implementation of EdDSA, the Edwards-curve Digital Signature Algorithm, and was designed by Daniel Bernstein et al [56]. This algorithm is a variant of the Schnorr signature algorithm and uses elliptic-curve cryptography. More specifically, Ed25519 uses a twisted Edwards curve birationally equivalent to the curve Curve25519, the Montgomery curve over the

prime field defined by the prime number $2^{255} - 19$. A digital signature is used to provide authenticity and integrity of a message. A valid digital signature provides the receiver of a message assurance that the message was created by the sender of the message and that the message was not altered in transit.

A Ed25519 keypair consists of a 64-byte private key and a 32-byte public key. At key generation, a 32-byte random seed, for instance the output of SHA256 on some random input, is generated. This seed is then hashed using SHA512, which expands the 32-byte seed to 64 bytes. This key is then split into a left half and a right half, both being 32 bytes long. The left half is the input for the Curve25519 function after a few operations on the first and last bytes of the left half. The resulting 32-byte secret scalar a and the 32-byte right half is the private key. The public key is generated by multiplying this secret scalar a by a generator B (the point $(x, 4/5)$), which results in a 32-byte group element A. Signature generation goes as follows: First of all, variable r is computed by hashing the message M and the right half of the private key using SHA512. Then, R is computed by multiplying r with the generator B, or: $R = rB$. Next, $S = (r + H(\underline{R}, \underline{A}, M)a)$ mod ℓ where $\underline{R}$ and $\underline{A}$ are compressed points. The signature for a message M consists of R and S. Verification goes as follows: The verifier knows $\underline{A}$, $\underline{R}$, $\underline{S}$ and M. It first parses $\underline{A}$, then it calculates H($\underline{\text{R}}$, $\underline{\text{A}}$, M). Then it checks the group equation $8SB = 8R + 8H(\underline{R}, \underline{A}, M)A$. If the parsing and check succeeds, the signature is verified. There are various implementations written in a variety of languages of the Ed25519 algorithm and some differ a bit regarding representation of the keypair and calculation of the signature [57]. Apart from standalone libraries, Ed25519 is being used in several protocols, operating systems and networks [58].

6 Models

We assume that the communication channels between the manufacturer's portal p and host h, and between host h and target t are untrusted. For various reasons, we have to consider the possibility that the communication can be eavesdropped or modified in some form. Next to that, since it has been shown that internal communication of a car can easily be compromised if one has gained physical access to it, so we have to take adequate measures to keep the transferred firmware from being read by others.

The communication between the portal and the host is done over an untrusted channel. The portal is a backend server of the manufacturer, the host is the TCU inside the vehicle to be updated, the target is the ECU to be updated. Distinct messages between the same sender and receiver set are required to have distinct nonces. For example, the lexicographically smaller public key can use odd increasing nonces, while the lexicographically larger public key uses even increasing nonces. The used nonces are long enough that randomly generated nonces have negligible risk of collision.

6.1 Key storage

Now we have outlined the protocols we will use for our model, we also need to store the public/private keys used for encryption. First of all, we need a public/private keypair for our portal, host and targets. These keys need to be stored safely on the devices that are used. At manufacturing time, each device should get its own keypair, with a redundant copy of the private key being placed in the ROM at the same time to prevent the key integrity attacks mentioned earlier in this thesis. Furthermore, the portal should have knowledge of all public keys of the host. Also, all hosts should have knowledge of the public key of the portal. Moreover, every target should have knowledge of the public key of the host and the host should have a list of the public keys of every distinct target in the vehicle.

6.2 Host to portal

There are two different options. One option is that the host h asks the portal p if there are any updates available for the ECUs of the car. The model is shown below in Figure 9. First, an authenticated ECDH key echange is executed between the host h and the portal p, initiated by the host. If this key exchange succeeds, the protocol will move on to the next step. There, the host asks the portal if there are any updates available for the ECUs it maintains. Therefore it needs a list of ECUs and their versions, which is being stored by the host. Formally, the portal asks the host for a version list and the host replies with the version list. The host encrypts the ECU version list with the secret key it shares with the portal and sends it. The portal then checks if it has any updates available and replies to the host with an encrypted response. Then,

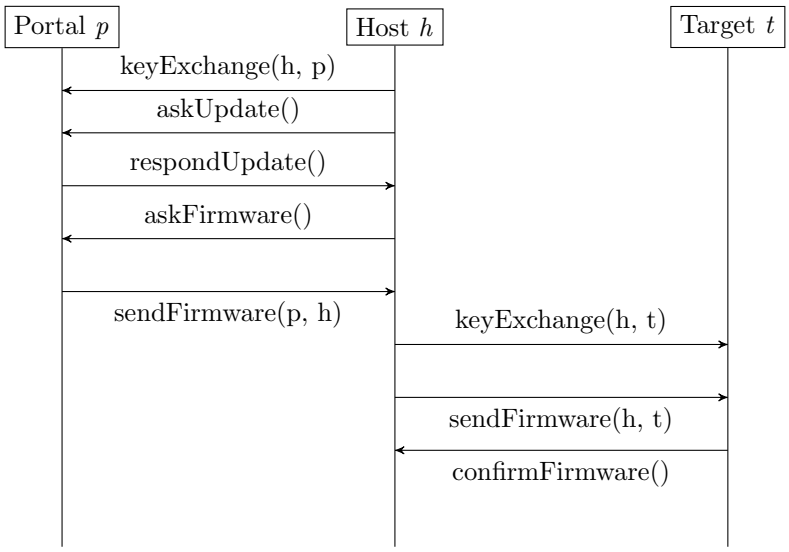

Figure 9: Model - Initiated communication by host h to the portal p. The firmware update is sent to target t. The used functions are explained in Section 7

if there are any updates, the host asks the portal for each of the corresponding updates. The portal encrypts and sends the firmware to the host. The host initiates an authenticated ECDH key exchange with the target ECU and sends the firmware to the ECUs. Finally, if the firmware update succeeds, the target confirms the new firmware version to the host.

6.3 Portal to host

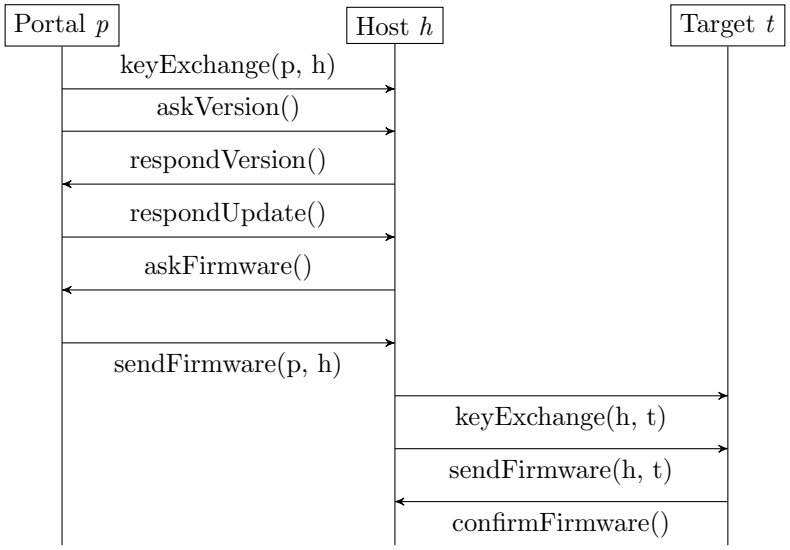

Figure 10: Model - Initiated communication by portal p to the host h. The firmware update is sent to target t. The used functions are explained in Section 7

The other option is that the portal asks the host to return the current versions of the ECUs and checks whether any ECU should receive a firmware update. This model is shown in Figure 10. First, an authenticated ECDH key echange is executed between the host and the portal, initiated by the portal. If this key exchange succeeds, the protocol will move on to the next step. There, the portal asks the host to send all the versions of the firmware of all ECUs in the system. The host encrypts the ECU version list with the shared secret key and sends it. The portal then checks if it has any updates available and replies to the host with an encrypted response. Then, if there are any updates, the host asks the portal for each of the corresponding updates. The portal encrypts and sends the firmware to the host. The host initiates an authenticated ECDH key exchange with the target ECU and sends the firmware to the target. Next, if the firmware update succeeds, the target confirms the new firmware versions to the host.

7 Definitions and Functions

The two proposed models that have been drafted in the previous section are defined on a high level. To define them on a lower level, we will need to specify more. First, we will list the different definitions and variables we are going to use for our functions. Then, we will specify the different functions that are used by the models.

7.1 Definitions

7.1.1 Keys

pkx. The public long-term key of party x, generated during manufacturing time.
skx. The secret long-term key of party x, generated during manufacturing time.
pky. The public long-term key of party y, generated during manufacturing time.
sky. The secret long-term key of party y, generated during manufacturing time.

$ecpkx$. The ephemeral elliptic-curve public key of party x.
$ecskx$. The ephemeral elliptic-curve secret key of party x.
$ecpky$. The ephemeral elliptic-curve public key of party y.
$ecsky$. The ephemeral elliptic-curve secret key of party y.

7.1.2 Variables

c. The ciphertext or encrypted message.
e. The encrypted payload.
k. The key k.
n. A string of random bytes used as a nonce.
m. The plaintext or message.
r. A string of random bytes.
s. The signature of a message using Ed25519.
ss. The shared secret generated from Curve25519 and shared between two parties.

$sign(x, y)$ = signs the message y with private key x using Ed25519.
$verify(x, y)$ = verifies the message y with public key x using Ed25519.
$encrypt(x, y, z)$. Encrypts and authenticates a message x with a nonce y using key z.
$authenticate(x, y, z)$. Authenticates and decrypts a message x with a nonce y using key z.
$pair(x, y) = (x, y)$. Pairs two variables.
$split(x, y) = x, y$. Split two variables.
$send(x, y)$ Sends data x to target y.
$receive(x, y)$ Receive data x from target y.

7.2 Functions

7.2.1 keyExchange(x,y)

The key exchange `keyExchange(`x`,`y`)` is an authenticated ECDH key exchange between initiator x and responder y. The initiator generates an elliptic-curve private key $ecskx$ for this specific session and calculates a public key $ecpkx$ from the secret key using Curve25519. It uses its own private key skx to sign the public key $ecpkx$ using Ed25519 resulting in s to be sent to responder y (Algorithm 1).

Now, the responder y verifies the signature of the public key. If verification of the public key succeeds, it generates an own elliptic-curve private key $ecpky$ for this specific session and calculates a public key $ecpky$ from the secret key using Curve25519. It can now generate the shared secret ss by using its own private key $ecsky$ and the received public key $ecpkx$. Then it uses its own private key sky to sign the public key $ecpky$ using Ed25519 resulting in s, which is sent to the initiator (Algorithm 2).

Then, the initiator x verifies the signature of the public key. It can now generate the shared secret ss by using its own private key $ecskx$ and the received public key $ecpky$ (Algorithm 3). The key exchange is now established. More formally, the following actions are taken:

Algorithm 1 ECDH Key Exchange algorithm (initiator x)

$ecskx \leftarrow r$
$ecpkx \leftarrow Curve25519(ecskx, 9)$
if $ecpkx == \perp$ **then**
  $s \leftarrow sign(skx, ecpkx)$
  if $s == \perp$ **then**
    $send(s, y)$
  end if
end if

If everything went as planned, both parties now share the same secret key ss. If not, the function returns an error and the process stops. Note that for readability purposes, we did not include any memory allocation operations in the listed algorithms. Also note that the functions we are going to use originally include the variable the resulting value will be put into when calling the function. Officially, these functions return 0 on success or -1 if an error occured. For readability purposes, in the provided algorithms we check for $\perp$ if a function succeeds, otherwise it has failed. Now that the `keyExchange(x,y)` has succeeded, we will define a set of functions that is used within all other functions. We define it once to not having to repeat ourselves and keeping things well-ordered.

Algorithm 2 ECDH Key Exchange algorithm (responder y)

$z \leftarrow verify(s, ecpkx, pkx)$
if $z == 1$ **then**
  $ecsky \leftarrow r$
  $ecpky \leftarrow Curve25519(ecsky, 9)$
  if $ecpky == \perp$ **then**
    $ss \leftarrow Curve25519(ecsky, ecpkx)$
    if $ss == \perp$ **then**
      $s \leftarrow sign(sky, ecpky)$
      if $s == \perp$ **then**
        $c \leftarrow pair(s, ecpky)$
        $send(c, x)$
      end if
    end if
  end if
end if

Algorithm 3 ECDH Key Exchange algorithm (initiator x (cont.))

$z \leftarrow verify(s, ecpkx, pky)$
if $z == \perp$ **then**
  $ss \leftarrow Curve25519(ecskx, ecpky)$
end if

7.2.2 pack(m)

The `pack` function encrypts, authenticates and packs the message m with a nonce n and a key k resulting in an authenticated ciphertext e. More specifically, it performs the following actions: First, it generates a nonce n from the string of random bytes r. It then uses the message m, the nonce n and the shared secret ss to create an authenticated ciphertext e. This function is shown in Algorithm 4. The Poly1305-XSalsa20 cipher suite is used for encryption and authentication.

Algorithm 4 Pack function

$n \leftarrow r$
$e \leftarrow encrypt(m, n, ss)$
if $e == \perp$ **then**
  $en \leftarrow pair(e, n)$
end if

7.2.3 unpack(en)

The `unpack` function unpacks, decrypts and validates en. This en consists of the pair (e,n). The pair is split into authenticated ciphertext e and nonce n. Next, it uses the shared secret ss, the nonce n and the authenticated ciphertext e to authenticate and decrypt the ciphertext e using Poly1305-XSalsa20, resulting in message m. More formally:

Algorithm 5 Unpack function

$e, n \leftarrow split(e, n)$
$m \leftarrow authenticate(e, n, ss)$

7.2.4 askUpdate()

The host h asks the portal p if there are any updates with `askUpdate()`. It will create a message m that asks the portal for an update. It will then `pack` message m and send the resulting payload em to the portal. The portal will unpack em, resulting in the original message m being recovered. Now, the portal can process message m. Message m will be a simple byte representation of the `askUpdate()` function, notated as {request}:

Algorithm 6 askUpdate() function (host)

1: $m \leftarrow \{request\}$
2: $em \leftarrow pack(m)$
3: **if** $em == \perp$ **then**
4: $\quad send(em, p)$
5: **end if**

Algorithm 7 askUpdate() function (portal)

1: $receive(em, h)$
2: $m \leftarrow unpack(em)$
3: **if** $m == \perp$ **then**
4: $\quad process(m)$
5: **end if**

7.2.5 askVersion()

The portal asks the host which versions of firmware the ECUs currently have with `askVersion()`. It will create a message m that asks the host for the current firmware versions. It will then `pack` message m and send the resulting payload em to the host. The host will unpack em, resulting in the original message m being recovered. Now, the host can process message m. Message m will be a simple byte representation of the `askVersion()` function, notated as {request}.

Algorithm 8 askVersion() function (portal)

1: $m \leftarrow \{request\}$
2: $em \leftarrow pack(m)$
3: **if** $em == \perp$ **then**
4: $\quad send(em, h)$
5: **end if**

Algorithm 9 askVersion() function (host)

1: $receive(em, p)$
2: $m \leftarrow unpack(em)$
3: **if** $m == \perp$ **then**
4: $\quad process(m)$
5: **end if**

7.2.6 respondVersion()

The host will respond to the portal with the versions of firmware the ECUs currently have `respondVersion()`. It will create a message m that asks the portal for an update. It will then `pack` message m and send the resulting payload em to the portal. The portal will unpack em, resulting in the original message m being recovered. Now, the portal can process message m. Message m will be a byte array representation of {`targetID`, `version`}. The `targetID` field will be four bytes long, the `version` field will be four bytes long. For instance, m will look like {{`0x46,0x5B`},{`0x03,0x8A`}}. This will allow for 256^2 different ECUs to be indexed with 256^2 different versions.

Algorithm 10 respondVersion() function (host)

1: $m \leftarrow \{targetID, version\}$
2: $em \leftarrow pack(m)$
3: **if** $em == \perp$ **then**
4: $\quad send(em, p)$
5: **end if**

Algorithm 11 respondVersion() function (portal)

1: $receive(em, h)$
2: $m \leftarrow unpack(em)$
3: **if** $m == \perp$ **then**
4: $\quad process(m)$
5: **end if**

7.2.7 respondUpdate()

The portal replies the host with the firmware updates that it has available for the host's ECUs `respondUpdate()`. It will create a message m with the reply. It will then `pack` message m and send the resulting payload em to the host. The host will unpack em, resulting in the original message m being recovered. Now, the host can process message m. Message m will be a byte array representation of the ECUs that have an available update on the portal, looking

like {`targetID`}.The `targetID` field will be four bytes long. For instance, m will look like {`0x46,0x5B`}.

Algorithm 12 respondUpdate() function (portal)

1: $m \leftarrow \{targetID\}$
2: $em \leftarrow pack(m)$
3: **if** $em == \perp$ **then**
4: $\quad send(em, p)$
5: **end if**

Algorithm 13 respondUpdate() function (host)

1: $receive(em, h)$
2: $m \leftarrow unpack(em)$
3: **if** $m == \perp$ **then**
4: $\quad process(m)$
5: **end if**

7.2.8 askFirmware()

Now that the host knows which targets can receive a firmware update, it will create a message m that asks the portal for these specific firmware binaries `askFirmware()` for every firmware version. It will `pack` message m and send the resulting em to the portal. The portal will unpack em, resulting in the original message m being recovered. Now, the portal can process message m. Message m will be a byte array representation of {`targetID`, `version`}. The `targetID` field will be four bytes long, the `version` field will be four bytes long. For instance, m will look like {{`0x46,0x5B`},{`0x03,0x8A`}}.

Algorithm 14 askFirmware() function (host)

1: $m \leftarrow \{targetID, version\}$
2: $em \leftarrow pack(m)$
3: **if** $em == \perp$ **then**
4: $\quad send(em, p)$
5: **end if**

Algorithm 15 askFirmware() function (portal)

1: $receive(em, h)$
2: $m \leftarrow unpack(em)$
3: **if** $m == \perp$ **then**
4: $\quad process(m)$
5: **end if**

7.2.9 sendFirmware(x, y)

The initiator x sends the firmware update to the responder y by calling the `sendFirmware(x, y)` function. It will create a message m with the reply. It will then `pack` message m and send the resulting payload em to the responder. The responder will unpack em, resulting in the original message m being recovered. Now, the responder can process message m. Message m will be a byte array representation of the ECUs that have an available update on the portal, looking like {`targetID`, `message`, `mlen`}.The `targetID` field will be four bytes long. The `message` field will be the container of the binary firmware, it can be of any arbitrary size ≥ 1. The length of the `message` field is saved in the `mlen` field.

Algorithm 16 sendFirmware(x,y) function (initiator)

1: $m \leftarrow \{targetID, message, mlen\}$
2: $em \leftarrow pack(m)$
3: **if** $em == \perp$ **then**
4: $\quad send(em, y)$
5: **end if**

Algorithm 17 sendFirmware(x,y) function (responder)

1: $receive(em, x)$
2: $m \leftarrow unpack(em)$
3: **if** $m == \perp$ **then**
4: $\quad process(m)$
5: **end if**

7.2.10 confirmFirmware()

After the firmware update process was succesful, the target ECU will confirm to the host that the update was succesful `confirmFirmware()`. It will `pack` message m and send the resulting e to the host. The host will unpack e, resulting in the original message m being recovered. Now, the host can process message m. Message m will be a byte array representation of {`targetID`, `version`}. The `targetID` field will be four bytes long, the `version` field will be four bytes long.

For instance, m will look like {{`0x46,0x5B`},{`0x03,0x8A`}}. The host updates its internal list of firmware versions and either calls another `askFirmware()` function, or if there are no updates left, does nothing.

Algorithm 18 confirmFirmware() function (target)

1: $m \leftarrow \{targetID, version\}$
2: $em \leftarrow pack(m)$
3: **if** $em == \perp$ **then**
4: $\quad send(em, h)$
5: **end if**

Algorithm 19 confirmFirmware() function (host)

1: $receive(em, t)$
2: $m \leftarrow unpack(em)$
3: **if** $m == \perp$ **then**
4: $\quad process(m)$
5: **end if**

8 Implementation

Our implementation on the ECU uses a modified version of the Networking and Cryptography library NaCl (pronounced as: salt) implementation by Bernstein, Lange and Schwabe [59, 60]. This cryptographic library offers various cryptographic primitives. More specifically, our implementation relies mostly on the AVR 8-bit implementation of NaCl, written by Schwabe and Hutter [61, 62] and it is optimized for microprocessors with 8-bit registers. Since the microprocessor we use is a 32-bit microprocessor with 32-bit registers, the code can be optimized for this specific processor. The NaCl library takes advantage of higher-level language features to simplify the APIs for those languages. For example:

- A message is represented in C NaCl as two variables: an array variable `m` and an integer variable `mlen`. Higher-level APIs (like the C++ one) use a single string variable `m` that knows its own length.
- The C NaCl functions return error codes to indicate errors. Programmers are still expected to check for these error codes and respond accordingly.
- The C NaCl functions write output strings via pointers. Higher-level APIs (like C++) return the strings as function values.

The implementation is written in the ANSI C language for portability, but also supports the C++ language. Usually, cryptographic libraries leave the choice of primitives to the programmer. Often programmers pass the choice along to users, which is generally a bad idea. NaCl encourages users to simply say "sign this message" or "encrypt this message". NaCl has a mechanism through which a cryptographer can easily specify the choice of signature system, without the programmer or user having to worry about it. NaCl avoids all data flow from secret information to the instruction pointer and the branch predictor. There are no conditional branches like *If statements* with conditions based on secret information and the code runs in constant time. This prevents certain timing attacks described in earlier sections, while preserving the claimed high speeds.

8.1 Proof of Concept

The proof of concept that has been implemented on the MPC5566 microprocessor uses other functions of the NaCl library. For our proof of concept, ANSI C is used, not the included C++ APIs. Not all features that come with NaCl are used in this implementation. More specifically, these three sets of functions are used:

- The Curve25519 ECDH key exchange uses the `crypto_scalarmult` function.
- The signing of the ECDH public keys is done with the `crypto_sign` function.

- The encryption of the communication is done through the `crypto_secretbox` function.

To provide the proof of concept with a working example, we have also included the following constants in the program:

- Keypairs for the ECU, TCU and portal,
- The update *to be applied to an ECU*,
- An ECU list with corresponding versions,
- A list of ECUs to be updated.

Since the proof of concept focuses on the secure communication process instead of the actual flashing of an ECU, we refrain from the following specific actions:

- The actual ECU flashing process (demo reflashes memory every time),
- A key exchange with multiple ECUs for different update versions (one is enough for demonstration),
- The processing of the messages sent and the order of the messages (only comparison of 2 bytes).

The proof of concept focuses on the simulation the process of the model shown in Figure 9. The implementation can be found on `https://github.com/remyspaan/ecu-update`. Appendix A explains how to test the proof of concept on a MPC5566 Evaluation Board, the MPC5566EVB, which was used to test the code on. In this proof of concept, all three entities in the update process (portal, host, target) are combined. There are also specific implementations for only the portal, host and target as not every part needs all the code. For instance, the target will only need to know how to handle the `keyExchange(x,y)`, `sendFirmware(x,y)` and `confirmFirmware()` functions.

Since the MPC5566 does not include any Electrically Erasable Programmable Read Only Memory (EEPROM), the randomization of the numbers used as nonce is done from a static seed instead of a seed that is updated every time on the FLASH. Since our build will reflash the memory before runtime anyways, implementing proper randomization would be impossible anyways. The benefit of having *static* random numbers improves the readability when debugging anyways. A PRNG would definitely be a required feature in a final version of the implementation. We will go through all of the aspects of the proof of concept now.

8.2 Memory allocation

The NaCl library claims that it does not use dynamic memory allocation for its cryptographic computations but it still should be usable in environments like our MPC, with limited storage

available [59]. However, we still need to allocate memory to make sure that the memory we want to fill with data can actually hold as much data as we want it to, before we want to use it. For that purpose, we use the `calloc` function, which zero-initializes the memory for an array of x elements. For instance, if we want to allocate memory for a message `m` with length `mlen` and fill it with the message `message`:

```
int i;
unsigned char *m;
unsigned char message[16] = {
  0x7f,0xc4,0xbd,0xfe,0x2c,0xc3,0x83,0x9e,0x2b,0x27,0x85,0x85,0xf3,0xef,0xe1,0x5e
};
mlen = sizeof(message);
m = (unsigned char*)calloc(mlen,1);
if(!m)
  fail("allocation of memory for m failed");
for (i=0;i<16;i++)
  m[i] = message[i];
```

The return value of the `calloc` function is a `void *` so we have to cast it to an `unsigned char *`. If the memory allocation fails, a null-pointer is returned. We can use this result to check whether the `calloc` succeeded. If not, we can at least fail safely. After using the memory we use the `free` command to free the allocated memory. In the tests that come with the NaCl implementation, 32 extra bytes are *callocated* for each allocated variable of the type `unsigned char*`. Next, the pointer's location is increased by 16 places so that the variable has 16 bytes of zero's before and after the actual value. If we want to use the `free` function to free up the used memory, we have to make sure that the pointer points to the start of the actual memory location again, so we do not free space that was not part of the pointer anyways.

8.3 Function and program constants

Next to the constants that are defined by NaCl, we have defined a few constants that are used as a message and checked for when a function is executed by the program. We chose for two bytes to have enough space available for a variety of functions, and to have space for different responses in the same function. To avoid confusion with the memory locations that should be zero-allocated, we have chosen to use hexidecimal representations above `0x00`:

```
const unsigned char FUNC_KEY_EXCHANGE[] = {0x01, 0x01};
const unsigned char FUNC_KEY_EXCHANGE_CONT[] = {0x01, 0x02};
const unsigned char FUNC_ASK_UPDATE[] = {0x02, 0x01};
const unsigned char FUNC_ASK_VERSION[] = {0x03, 0x01};
```

```
const unsigned char FUNC_RESPOND_VERSION[] = {0x04, 0x01};
const unsigned char FUNC_RESPOND_UPDATE[] = {0x05, 0x01};
const unsigned char FUNC_ASK_FIRMWARE[] = {0x06, 0x01};
const unsigned char FUNC_SEND_FIRMWARE[] = {0x07, 0x01};
const unsigned char FUNC_CONFIRM_FIRMWARE[] = {0x08, 0x01};
```

8.4 Key generation and exchange

The portal and every ECU and TCU have a long-term public/private keypair. This keypair is generated during manufacturing time. Since this key will be used for the mutual authentication process between two parties, all TCUs have knowledge of the public key of the portal and all ECUs have knowledge of the public key of its TCU. Likewise, the portal has knowledge of the public key of all TCUs and the TCU has knowledge of the public key of all its ECUs.

Since we will be using Ed25519 for signing the public keys in the Curve25519 key exchange, the long-term keys stored on the devices will be Ed25519 keys. We have seen that a Ed25519 keypair can be deduced from a 32-byte seed so only storing the 32-byte seed is an option. However, for both computational and practical issues, we first chose to store both the private seed (32-byte) and the public (32-byte) key on the devices and the portal. In this way we do not need to calculate the keypair every time it is needed for computation.

It has been mentioned that some of the Ed25519 implementations, including the one that comes with NaCl, differ from the algorithm description in the original paper [56]. The website of NaCl even mentions that the current version of Ed25519 is still a prototype and will be replaced in a future release [63]. The `crypto_sign` function does not require the public key during the calculation of the hash. So insteaed of $H(\underline{R}, \underline{A}, M)$, $H(\underline{R}, M)$ is calculated. The public key is only used for verifying a signature and this means that a device or the portal does not need to store its own public key as it never uses it. So we end up only storing the private seed (32-byte) on the device and the portal. A public key will still be deduced from the private key at manufacturing time, only to store it on the devices that need knowledge of this public key.

8.5 crypto_secretbox

Our implementation relies on the `crypto_secretbox` function that comes with the NaCl library. This function ensures secret-key authenticated encryption and is designed to meet the standard notions of privacy and authenticity for secret-key authenticated-encryption using nonces [64]. The secret key that is used in this function is the shared secret key that the two parties have agreed upon during the initial key exchange. The `crypto_secretbox` function is really a call to the `crypto_secretbox_xsalsa20poly1305` function. The used cryptographic primitives are

XSalsa20 (which is a variant of Salsa20 using a 192-bit nonce instead of a 64-bit nonce) and Poly1305 [65]. These primitives have been described in Section 5.6. This function basically takes care of all the encryption and authentication needed for our model, except the initial key exchange. A call to the `crypto_secretbox` function in C requires the following parameters:

```
const unsigned char k[crypto_secretbox_KEYBYTES];
const unsigned char n[crypto_secretbox_NONCEBYTES];
const unsigned char m[...];
unsigned long long mlen;
unsigned char c[...];
unsigned long long clen;
```

And is called as follows:

```
crypto_secretbox(c,m,mlen,n,k);
```

The `crypto_secretbox` function encrypts and authenticates a message `m[0], m[1], ..., m[mlen-1]` using a secret key `k[0], ..., k[crypto_secretbox_KEYBYTES-1]` and a nonce `n[0], n[1], ..., n[crypto_secretbox_NONCEBYTES-1]`. The `crypto_secretbox` function puts the ciphertext into `c[0], c[1], ..., c[mlen-1]` and the length of the cipthertext in `clen`. It then returns 0. If there is any error during execution of the `crypto_secretbox` function, it returns -1. Decrypting a message goes as follows:

```
crypto_secretbox_open(m,c,clen,n,k);
```

The `crypto_secretbox_open` function verifies and decrypts a ciphertext `c[0], c[1], ..., c[clen-1]` using a secret key `k[0], k[1], ..., k[crypto_secretbox_KEYBYTES-1]` and a nonce `n[0], ..., n[crypto_secretbox_NONCEBYTES-1]`. The `crypto_secretbox_open` function puts the plaintext into `m[0], m[1], ..., m[clen-1]`. It then returns 0. If the ciphertext fails verification at any point, the `crypto_secretbox_open` instead returns -1 [66].

The `crypto_secretbox` function has another important requirement. According to the documentation, the first `crypto_secretbox_ZEROBYTES` (in case of encryption) or the first `crypto_secretbox_BOXZEROBYTES` (in case of decryption) should be all 0. In our case, the `crypto_secretbox_xsalsa20poly1305` function requires the first 16 bytes of the message `m` and the first 32 bytes of the ciphertext `c` to be 0. Both `crypto_secretbox` and `crypto_secretbox_open` ensure that the first 16 or 32 bytes of `m` and `c` are 0. To do that we have to just initialise the variables with 16 or 32 extra bytes and skip these bytes before we start inserting data.

8.6 crypto_scalarmult

The `crypto_scalarmult` function does scalar multiplication and is used in the Curve25519 function. This is because the selected primitive for this function is the `crypto_scalarmult_curve25519` function [66]. It also is the only available primitive for this function for now in the NaCl implementation. There are two main functions: `crypto_scalarmult(q,n,p)` and `crypto_scalarmult_base(q,n)`. The first function uses `n` and `p` as input and puts the result in `q`. The second function uses `n` and 9 as a base (see Figure 8) and puts the resulting scalar into `q`.

8.7 crypto_sign

The `crypto_sign` or `crypto_sign_edwards25519sha512batch` or more specifically, the Ed25519 signature function signs any message `m` using a private key `sk`. A signed message `sm` can be verified by using the `crypto_sign_open` function, together with a public key `pk` that corresponds with the earlier used secret key. The maximum possible length `smlen` is `mlen+crypto_sign_BYTES`. The caller must allocate at least `mlen+crypto_sign_BYTES` (a constant value that should be defined for the corresponding primitive) for `sm`.

8.8 Options and optimizations

To prevent *optimizations* by the compiler, all optimizations have been turned off for the specific project and target. There are generally two options to prevent the compiler from eliminating code it does not deem necessary, like memory deallocations and checks if pointers are set. The first option is to disable all optimizations. Another option is to define pointers to secret data as *volatile* [67]. As shown in Figure 11, the compiler can optimize the code for speed or size. We tried several combinations of optimizations for our proof of concept, though none of them worked correctly after recompiling with options enabled.

To speed up the actual program when debugging, we found some code that cuts in time the speed of the program considerably. Therefore, before the crypto code (`main.cpp`) is ran, the `HW_init()` function is called. The `HW_init()` function initializes the hardware by setting the highest possible clock speed available for the MPC5566. This is because the JTAG debugger only lets us select speeds up to 100Mhz, while the MPC5566 supports 132Mhz. Additionally, this function disables the watchdog on the e200z6 core, though disabling the watchdog also disables the ability of the MPC to recover from errors or unexpected events [68].

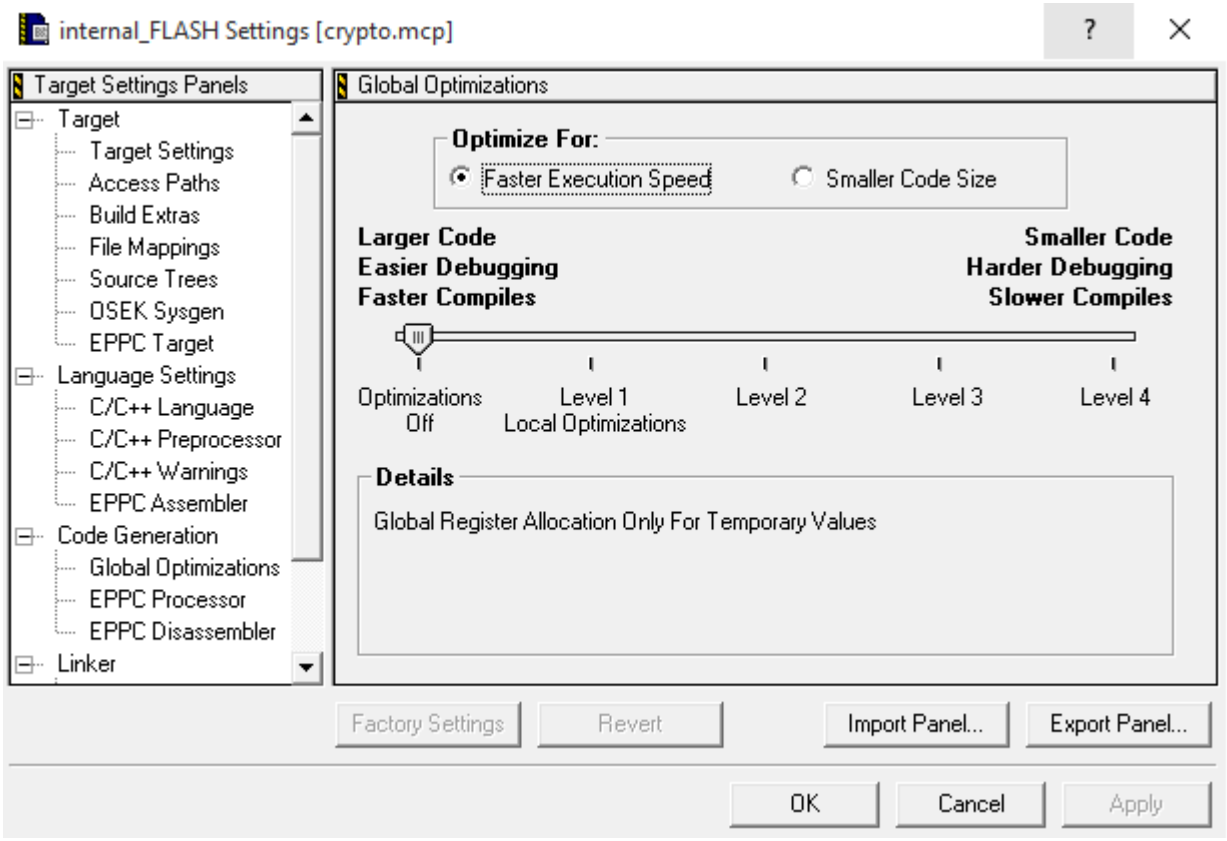

Figure 11: MPC Compiler Optimizations all turned off for this project and target.

9 Conclusions and future work

The purpose of this thesis was to identify the current shortcomings regarding automotive security. Additionally, this thesis provides a proof of concept for a system to secure a part of the update system of electronic control units in car systems. This proof-of-concept system ensures aspects like confidentiality, authenticity and integrity of a supplied update, while preventing common security pitfalls like man-in-the-middle attacks, timing attacks and replay attacks. The solution uses cryptographic primitives that are proven to be secure against all but quantum computer attacks and still is fast and small enough to be used in the field of car systems that are constrained by area, speed and temperature. This thesis focuses on specifying the protocols that are used where other studies on over-the-air updates assume that these protocols are secure and implemented correctly.

This study has taken a step towards the direction of making over-the-air updates for car systems more secure. Although this solution does not implement the entirety of creating, distributing, installing and using firmware updates for ECUs, using the resulting model does implement this specific part of security of an update. Where the security of a system always depends on the weakest link, efforts should be made to secure the other parts of the update system equally using other studies and proven secure methods. The implementation of this update system is for one specific but widely used ECU only, although the implementation is portable to other ECUs and TCUs while preserving required security properties and taking into account the different computational constraints.

Vehicle-to-vehicle (V2V) communication is not covered in this study, as the thesis' subject is applicable rather to communication between a car and a backend system than between cars itself. An interesting thing to investigate would be over-the-air distribution of firmware updates from vehicles to equivalent brands and types of vehicles in the vicinity. However, this addition will broaden the attack surface even more. Likewise, adding and implementing non-repudiation to the required security properties and the resulting solution should be considered for reading crash data (like a black box in a plane) or to prove that the latest ECU firmware versions were present in case of legal issues. Furthermore, private-key protection for this family of ECUs has not been proven thus it remains unclear whether an essential part of this implementation, the safety of the private key, can be guaranteed. A thorough side-channel analysis of this ECU should be executed before claiming the security of the private key.

References

[1] R. N. Charette, "This Car Runs on Code." www.spectrum.ieee.org/feb09/7649, 2009. [Online; accessed 27-September-2015]. 1, 8

[2] A. Brisbourne, "Teslas Over-the-Air Fix: Best Example Yet of the Internet of Things?." http://www.wired.com/2014/02/teslas-air-fix-best-example-yet-internet-things/, 2014. [Online; accessed 11-May-2015]. 1

[3] C. Thomas, "The race to go online in your car." http://www.telegraph.co.uk/motoring/news/11028324/The-race-to-go-online-in-your-car.html, 2014. [Online; accessed 11-May-2015]. 1

[4] GlobalCarsBrands, "Top 10 Newest Car Technologies That Have Revolutionized the Auto Industry." http://www.globalcarsbrands.com/top-10-newest-car-technologies-that-have-revolutionized-the-auto-industry/, 2015. [Online; accessed 6-May-2015]. 2

[5] International Organization for Standardization, "ISO 11898-1:2003 - Road vehicles – Controller area network (CAN)." http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=33422, 2009. [Online; accessed 11-May-2015]. 2

[6] U. E. Larson and D. K. Nilsson, "Securing vehicles against cyber attacks," in *Proceedings of the 4th annual workshop on Cyber security and information intelligence research: developing strategies to meet the cyber security and information intelligence challenges ahead*, p. 30, ACM, 2008. 5

[7] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, *et al.*, "Experimental security analysis of a modern automobile," in *Security and Privacy (SP), 2010 IEEE Symposium on*, pp. 447–462, IEEE, 2010. 5, 11

[8] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno, *et al.*, "Comprehensive experimental analyses of automotive attack surfaces.," in *USENIX Security Symposium*, San Francisco, 2011. 6, 11

[9] Bosch Automotive Service Solutions Inc., "Ford VCM II." http://www.boschdiagnostics.com/pro/products/ford-vcm-ii, 2015. [Online; accessed 26-November-2015]. 6

[10] C. Miller and C. Valasek, "Adventures in Automotive Networks and Control Units." http://www.ioactive.com/pdfs/IOActive_Adventures_in_Automotive_Networks_and_Control_Units.pdf, 2014. [Online; accessed 09-September-2015]. 8

[11] J. Pagliery, "Chrysler recalls 1.4 million hackable cars - Jul. 24, 2015." http://money.cnn.com/2015/07/24/technology/chrysler-hack-recall/, 2015. [Online; accessed 15-September-2015]. 9

[12] C. Miller and C. Valasek, "Remote Exploitation of an Unaltered Passenger Vehicle." http://www.ioactive.com/pdfs/IOActive_Remote_Car_Hacking.pdf, 2015. [Online; accessed 05-June-2015]. 9, 10

[13] T. Stevens, "GM issues fix for OnStar hack." http://www.cnet.com/news/ownstar-onstar-hack/, 2015. [Online; accessed 03-September-2015]. 10

[14] S. Kamkar, "OwnStar: Drive It Like You Hacked It." http://samy.pl/defcon2015/2015-defcon.pdf, 2015. [Online; accessed 27-October-2015]. 10

[15] A. Greenberg, "GM Took 5 Years to Fix a Full-Takeover Hack in Millions of OnStar Cars." http://www.wired.com/2015/09/gm-took-5-years-fix-full-takeover-hack-millions-onstar-cars/, 2015. [Online; accessed 26-October-2015]. 11

[16] N. Dhanjani, "Cursory Evaluation of the Tesla Model S: We Cant́ Protect Our Cars Like We Protect Our Workstations." http://www.dhanjani.com/blog/2014/03/curosry-evaluation-of-the-tesla-model-s-we-cant-protect-our-cars-like-we-protect-our-workstat html, 2014. [Online; accessed 05-June-2015]. 11

[17] S. Anthony, "Tesla's Model S can be located, unlocked, and burglarized with a simple hack." http://www.extremetech.com/extreme/179556-teslas-model-s-can-be-located-unlocked-and-burglarized-with-a-simple-hack, 2014. [Online; accessed 05-June-2015]. 11

[18] D. Spaar, "Beemer, Open Thyself! Security vulnerabilities in BMW's ConnectedDrive." http://www.heise.de/ct/artikel/Beemer-Open-Thyself-Security-vulnerabilities-in-BMW-s-ConnectedDrive-2540957.html, 2015. [Online; accessed 11-June-2015]. 12, 13

[19] I. Foster, A. Prudhomme, K. Koscher, and S. Savage, "Fast and vulnerable: a story of telematic failures," in *Proceedings of the 9th USENIX Conference on Offensive Technologies*, pp. 15–15, USENIX Association, 2015. 13, 14, 15

[20] D. K. Nilsson and U. E. Larson, "Secure firmware updates over the air in intelligent vehicles," in *Communications Workshops, 2008. ICC Workshops' 08. IEEE International Conference on*, pp. 380–384, IEEE, 2008. 16

[21] D. K. Nilsson, L. Sun, and T. Nakajima, "A framework for self-verification of firmware updates over the air in vehicle ECUs," in *GLOBECOM Workshops, 2008 IEEE*, pp. 1–5, IEEE, 2008. 16

[22] W. Kanda, Y. Yumura, Y. Kinebuchi, K. Makijima, and T. Nakajima, "Spumone: Lightweight cpu virtualization layer for embedded systems," in *Embedded and Ubiquitous Computing, 2008. EUC'08. IEEE/IFIP International Conference on*, vol. 1, pp. 144–151, IEEE, 2008. 16

[23] M. S. Idrees, H. Schweppe, Y. Roudier, M. Wolf, D. Scheuermann, and O. Henniger, "Secure automotive on-board protocols: a case of over-the-air firmware updates," in *Communication Technologies for Vehicles*, pp. 224–238, Springer, 2011. 16

[24] I. Studnia, V. Nicomette, E. Alata, Y. Deswarte, M. Kaâniche, and Y. Laarouchi, "Survey on security threats and protection mechanisms in embedded automotive networks," in *Dependable Systems and Networks Workshop (DSN-W), 2013 43rd Annual IEEE/IFIP Conference on*, pp. 1–12, IEEE, 2013. 17

[25] EVITA, "EVITA Factsheet." http://evita-project.org/EVITA_factsheet.pdf, 2011. [Online; accessed 29-November-2015]. 18

[26] B. Weyl, M. Wolf, F. Zweers, T. Gendrullis, M. S. Idrees, Y. Roudier, H. Schweppe, H. Platzdasch, R. El Khayari, O. Henniger, D. Scheuermann, A. Fuchs, L. Apvrille, G. Pedroza, H. Seudie, J. Shokrollahi, and A. Keil, "EVITA Deliverable D3.2, Secure On-Board Architecture Specification - Version 1.3." http://evita-project.org/Deliverables/EVITAD3.2.pdf, 2011. [Online; accessed 28-October-2015]. 18, 19

[27] H. Schweppe, S. Idrees, Y. Roudier, B. Weyl, R. El Khayari, O. Henniger, D. Scheuermann, G. Pedroza, L. Apvrille, H. Seudie, H. Platzdasch, M. Sall, A. Keil, and M. Wolf, "EVITA Deliverable D3.3, Secure Secure On-Board Protocols Specification - Version 1.4." http://evita-project.org/Deliverables/EVITAD3.3.pdf, 2011. [Online; accessed 28-October-2015]. 18

[28] HIS, "HIS AK Security Version 1.8.2." http://portal.automotive-his.de/images/pdf/FlashProgramming/lh_his_freischaltung_ver1_8_2.pdf, 2010. [Online; accessed 15-October-2015]. 19

[29] D. Dolev and A. C. Yao, "On the security of public key protocols," *Information Theory, IEEE Transactions on*, vol. 29, no. 2, pp. 198–208, 1983. 21

[30] BimmerBoost, "Real BMW S55, N63, N63TU, S63TU tuning coming - F-Series Infineon TriCore ECU's cracked." http://www.bimmerboost.com/content.php?5514-Real-BMW-S55-N63-N63TU-S63TU-tuning-coming-F-Series-Infineon-TriCore-ECU-s-crackedf, 2014. [Online; accessed 28-October-2015]. 22, 27

[31] Anthony, Sebastian, "The first rule of zero-days is no one talks about zero-days (so well explain)." http://arstechnica.com/security/2015/10/the-rise-of-the-zero-day-market/, 2015. [Online; accessed 30-November-2015]. 23

[32] R. Moore, *Cybercrime: Investigating high-technology computer crime*. Routledge, 2010. 23

[33] A. I. Awad, A. E. Hassanien, and K. Baba, "Advances in security of information and communication networks," p. 211, 2013. 28

[34] M. Bishop, M. Dilger, *et al.*, "Checking for race conditions in file accesses," *Computing systems*, vol. 2, no. 2, pp. 131–152, 1996. 28

[35] J. J. Savard, "The Ideal Cipher." http://www.quadibloc.com/crypto/mi0611.htm, 1999. [Online; accessed 04-November-2015]. 28

[36] E. Uner, "Generating random numbers — Embedded." http://www.embedded.com/design/configurable-systems/4024972/Generating-random-numbers, 2004. [Online; accessed 9-November-2015]. 28

[37] S. Buntz, T. Hogenmuller, S. Korzin, K. Matheus, M. Mehnert, N. Morand, T. Streichert, M. Tazebay, J. Wuelfing, and H. Zinner, "Tutorial for Lifetime Requirements and Physical Testing of Automotive Electronic Control Units (ECUs)." http://grouper.ieee.org/groups/802/3/RTPGE/public/july12/hoganmuller_01a_0712.pdf, 2012. [Online; accessed 11-November-2015]. 31

[38] Freescale Semiconductor, "MPC5566: 32-bit MCU for Automotive Powertrain Applications." http://www.freescale.com/products/power-architecture-processors/mpc5xxx-5xxx-32-bit-mcus/mpc55xx-mcus/32-bit-mcu-for-automotive-powertrain-applications:MPC5566, 2014. [Online; accessed 26-November-2015]. 31

[39] L. Mearian, "Ethernet is coming to cars." http://www.computerworld.com/article/2836400/ethernet-is-coming-to-cars.html, 2014. [Online; accessed 10-November-2015]. 32

[40] L. Jaks, "Security Evaluation of the Electronic Control Unit Software Update Process." https://drive.google.com/file/d/0B3qdCc__t5EybWxyVk5JTklhTVE/, 2014. 32

[41] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "Recommendation for Key Management Part 1: General (Revision 3)." http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general, 2012. [Online; accessed 21-September-2015]. 33

[42] K. Maletsky, "RSA vs ECC Comparison for Embedded Systems." http://www.atmel.com/Images/Atmel-8951-CryptoAuth-RSA-ECC-Comparison-Embedded-Systems-WhitePaper.pdf, 2015. [Online; accessed 10-September-2015]. 33

[43] X. Wang and H. Yu, "How to break MD5 and other hash functions," in *Advances in Cryptology–EUROCRYPT 2005*, pp. 19–35, Springer, 2005. 34

[44] V. Rijmen and E. Oswald, "Update on SHA-1." Cryptology ePrint Archive, Report 2005/010, 2005. 34

[45] NIST, "Federal Information Processing Standards Publication 180-4." http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf, 2015. [Online; accessed 13-November-2015]. 34

[46] NIST, "SHS Validation List." http://csrc.nist.gov/groups/STM/cavp/documents/shs/shaval.htm, 2015. [Online; accessed 19-November-2015]. 34

[47] D. J. Bernstein, "Curve25519: new Diffie-Hellman speed records," in *Public Key Cryptography-PKC 2006*, pp. 207–228, Springer, 2006. 35

[48] D. J. Bernstein and T. Lange, "SafeCurves: choosing safe curves for elliptic-curve cryptography." http://safecurves.cr.yp.to, 2014. [Online; accessed 14-October-2015]. 35

[49] Bernstein, Daniel J., "eSTREAM: the ECRYPT Stream Cipher Project." https://competitions.cr.yp.to/estream.html, 2014. [Online; accessed 16-January-2016]. 35

[50] D. J. Bernstein, "The Salsa20 family of stream ciphers," in *New stream cipher designs*, pp. 84–97, Springer, 2008. 35

[51] Bernstein, Daniel J., "Stream-cipher timings." http://cr.yp.to/streamciphers/timings.html, 2005. [Online; accessed 27-November-2015]. 35

[52] D. J. Bernstein, "The poly1305-aes message-authentication code," in *Fast Software Encryption*, pp. 32–49, Springer, 2005. 36

[53] Y. Nir and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols." https://tools.ietf.org/html/rfc7539, 2015. [Online; accessed 20-November-2015]. 36

[54] Bernstein, Daniel J., "ChaCha20 and Poly1305 in OpenSSH." http://blog.djm.net.au/2013/11/chacha20-and-poly1305-in-openssh.html, 2013. [Online; accessed 16-January-2016]. 36

[55] D. J. Bernstein, "Poly-1305 AES - A state-of-the-art message-authentication code." http://cr.yp.to/mac.html, 2005. [Online; accessed 19-November-2015]. 36

[56] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," *Journal of Cryptographic Engineering*, vol. 2, no. 2, pp. 77–89, 2012. 36, 56

[57] Warner, Brian, "Ed25519 Keys." https://blog.mozilla.org/warner/2011/11/29/ed25519-keys/, 2015. [Online; accessed 16-December-2015]. 37

[58] IANIX, "Things that use Ed25519." http://ianix.com/pub/ed25519-deployment.html, 2016. [Online; accessed 5-January-2016]. 37

[59] D. J. Bernstein, T. Lange, and P. Schwabe, "NaCl: Networking and Cryptography library." http://nacl.cr.yp.to/index.html, 2013. [Online; accessed 5-August-2015]. 53, 55

[60] D. J. Bernstein, T. Lange, and P. Schwabe, "The security impact of a new cryptographic library," in *Progress in Cryptology–LATINCRYPT 2012*, pp. 159–176, Springer, 2012. 53

[61] M. Hutter and P. Schwabe, "μNaCl The Networking and Cryptography library for microcontrollers." http://munacl.cryptojedi.org/index.shtml, 2013. [Online; accessed 11-August-2015]. 53

[62] M. Hutter and P. Schwabe, "NaCl on 8-Bit AVR Microcontrollers.," in *AFRICACRYPT*, pp. 156–172, Springer, 2013. 53

[63] Bernstein, Daniel J., "Signatures: crypto_sign." http://nacl.cr.yp.to/sign.html, 2012. [Online; accessed 5-January-2016]. 56

[64] J. H. An, "Authenticated encryption in the public-key setting: Security notions and analyses." Cryptology ePrint Archive, Report 2001/079, 2001. http://eprint.iacr.org/. 56

[65] Bernstein, Daniel J., "Extending the Salsa20 nonce." https://cr.yp.to/snuffle/xsalsa-20081128.pdf, 2011. [Online; accessed 25-May-2016]. 57

[66] D. J. Bernstein, "Cryptography in NaCl," *Networking and Cryptography library*, 2009. 57, 58

[67] Various Authors, "Coding rules - Cryptography Coding Standards." https://cryptocoding.net/index.php/Coding_rules, 2014. [Online; accessed 28-April-2016]. 58

[68] Terry, Bill, "MPC5500 Watchdog Timer - Configuration and Operation." http://cache.freescale.com/files/32bit/doc/app_note/AN2817.pdf, 2005. [Online; accessed 28-April-2016]. 58

[69] Sogeti Netherlands, "Organisatiestructuur Sogeti Nederland B.V. — juli 2015." https://www.sogeti.nl/sites/default/files/media/Organogram%20Sogeti%20juli%202015.pdf, 2015. [Online; accessed 29-November-2015]. 71

Appendices

A Testbed

To test the proof-of-concept implementation, one should have the two devices shown in Figure 12.

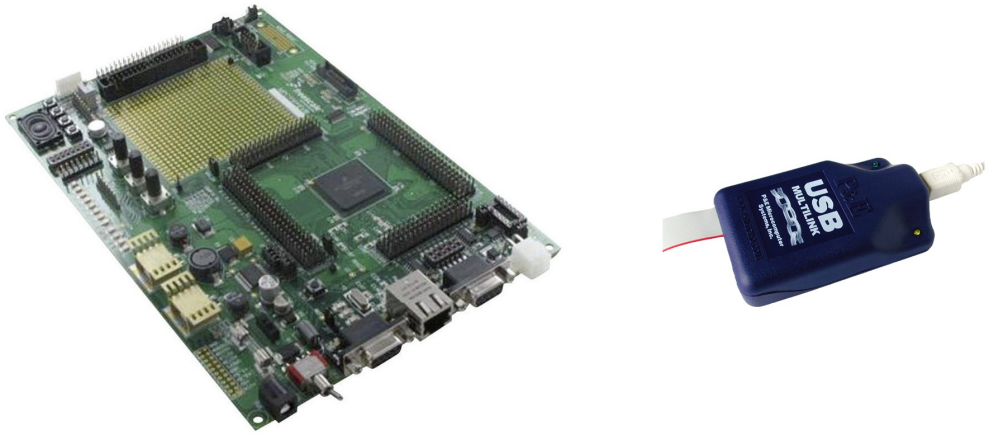

Figure 12: The two devices used for testing, the MPC5566EVB on the left and the USB Qorivva Multilink Interface on the right.

The proof-of-concept implementation can be tested by following these steps listed:

1. Download the MPC crypto code from the GitHub repository at `https://github.com/remyspaan/ecu-update`.
2. Download and install the latest CodeWarrior Development Studio for MPC55xx/MPC56xx from `http://www.nxp.com`.
3. Download the latest drivers for the USB Qorivva Multilink interface from `http://www.pemicro.com/products/product_viewDetails.cfm?product_id=15320021`.
4. Start the CodeWarrior IDE. At the startup dialog click *Load Previous Project* and load `crypto.mcp` into the IDE.
5. Set the target to `internal_FLASH` as shown in Figure 13.
6. Compile and build the code.
7. Select debug, select the appropriate connection port, interface type and target CPU.
8. Start the program with Connect (Reset).
9. The program will now erase and rewrite the flash.

10. When the program is started, enter `gotil main` in the status window of the debugger and press return. You will end up at the first line of the `main` of the program.

11. Debug through the code using the step icons in the tool bar or the shortcuts.

12. Set a breakpoint on the last command in the `main.cpp` file, the variable should return 0 after a successful attempt.

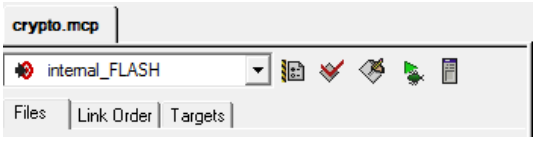

Figure 13: CodeWarrior IDE building to internal_FLASH.

B Sogeti Netherlands Organogram

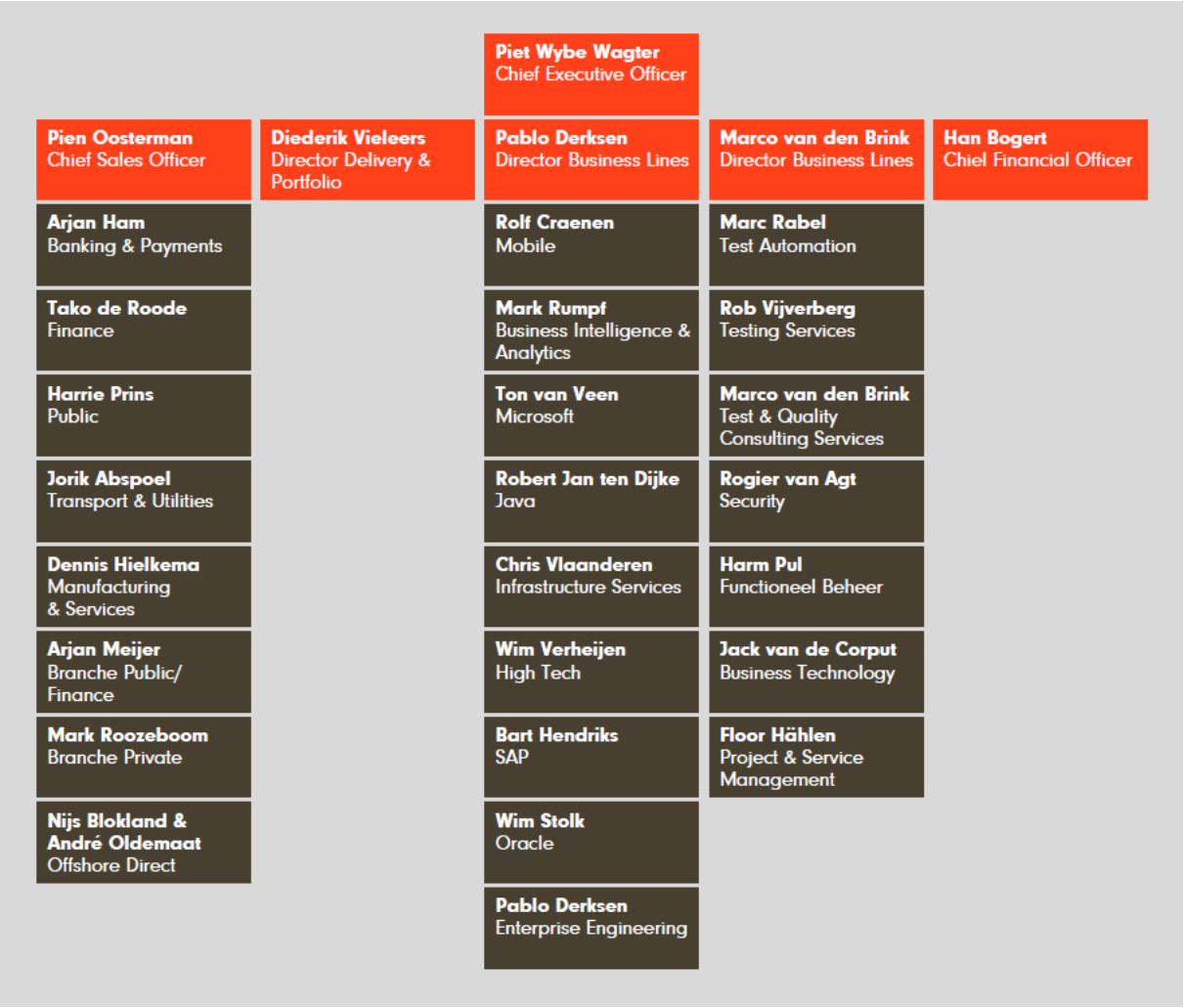

Figure 14: Sogeti Netherlands Organogram July 2015. Source: [69]