Peter Michael Hohl, BSc.

BECAUSE: Building an Efficient Compromise-Resilient Automotive Update System with End-to-End Security

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Dipl.-Ing. Michael Krisper BSc
Dipl.-Ing. Dr.techn. Georg Macher BSc

Institute for Technical Informatics
Head: Univ.-Prof. Dipl.-Inform. Dr.sc.ETH Kay Uwe Römer

Graz, November 2020

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date

Signature

Thanks

First, I want to thank Dipl.-Ing. Michael Krisper BSc and Dipl.-Ing. Dr.techn. Georg Macher BSc for the opportunity to develop this diploma thesis at the Institute for Technical Informatics.
Thank you for your advice and your qualified support during the process.

Special thanks to my parents who made my education and studies possible and always stood by my side.

Abstract

Automotive systems have experienced a dramatic increase in software functionality, paired with the addition of numerous electronic interfaces to the outside world in the last few years. This trend significantly widens the attack surface on cyber-physical systems in an automotive system, which human life directly depends on. Automotive update systems are the logical consequence and will soon be legally required in several countries.

Prior automotive update systems either ignore over-arching software dependencies of electronic components, do not consider the whole life-cycle of these electronic components or lack the functionality of attestation of proper update installation.

The proposed system BECAUSE addresses these issues while providing an efficient, highly compromise-resilient update solution with minimal impact on functional safety and diagnostic functionality. To demonstrate its feasibility, the proposed system was implemented on hardware using the upcoming RISC-V instruction set architecture. Measurements show that the proposed system is practicable and more efficient than current update systems in terms of metadata overhead and therefore update distribution.

Keywords: Automotive, Update, OTA, Bootloader, Firmware Update, Resilience, Cyber-Security

Contents

List of Figures

1 INTRODUCTION

The automotive industry originated in the trade of mechanical engineering. The first cars were purely mechanical and had little to no electrical or electronic parts. However, over the decades, electric and electronic components became more and more prevalent in the domain, and with that, also new challenges arose.
While the automotive industry traditionally is very experienced in ensuring safety in mechanical parts and physical materials, the new software based approaches and challenges are not that well known and researched.
Software may contain bugs which must be fixed even after the end-of-line at the production site, which is traditionally the last chance for correcting errors. But software needs to be updated even afterwards, and this is done via software updates. In the past, such updates were only possible with proprietary equipment and physical access to the vehicle and its electronic components, see [Steger et al., 2018] and [Shanmugam, 2014].
With the upcoming trend of cyber-physical systems in the last decade, vehicles became more and more connected. This permanent connectivity finally allows for updating the software of road vehicles without having to drive to a workshop, but update it "over-the-air" (OTA). Nevertheless, having the ability to update vehicles remotely also introduces a huge cyber-security risk and widens the attack surface of vehicles significantly.
In the automotive domain, human life directly depends upon the correct execution of the software. Security incidents occur every day, with sometimes devastating effects, see [*AutoThreat Intelligence Cyber Incident Repository*]. Patching software vulnerabilities via secure automotive update systems is therefore vital for the security of automotive systems and soon may even be legal requirements in various countries.

This master thesis introduces a lightweight, compromise-resilient update system with end-to-end integrity and attestation of proper update installation. The feasibility of this approach is demonstrated via a reference implementation of the update system.

1.1 What is an Automotive System

An automotive system is the entirety of a vehicle which is powered by a self-contained energy source and some kind of converter which utilizes this energy to propel the vehicle. Commonly known examples for automotive are cars, trucks, buses and motorcycles.

An automotive system typically consists of chassis, engine, powertrain, gearbox/transmission, axles, steering, brakes, wheels, but also many electric and electronic parts like lights, indicators, heating, air-condition, control panels, displays, infotainment systems, driver assistance systems with all its sensors, and many more.

1.2 Automotive Context

The first industrially manufactured vehicles had only a few electronic components. A prominent example, the first Model T by Ford Motor Company, only had an electronic ignition system. In 1919, Ford added electric lights as well as a starter and a generator, see [Ford Motor Company, 1919] for more information.
Over the years more and more electronic modules were added to road-vehicles. Famous examples are airbag-systems, anti-lock braking systems, electronic stability programs, tire-pressure monitoring systems, and parking assistant systems.

These electric modules are commonly referred to as electronic control units (ECU), see chapter 2.2. ECUs are interconnected through layers of various automotive bus-systems, see chapter 2.3, and constantly exchange data. Each additional electronic component makes the overall system more complex

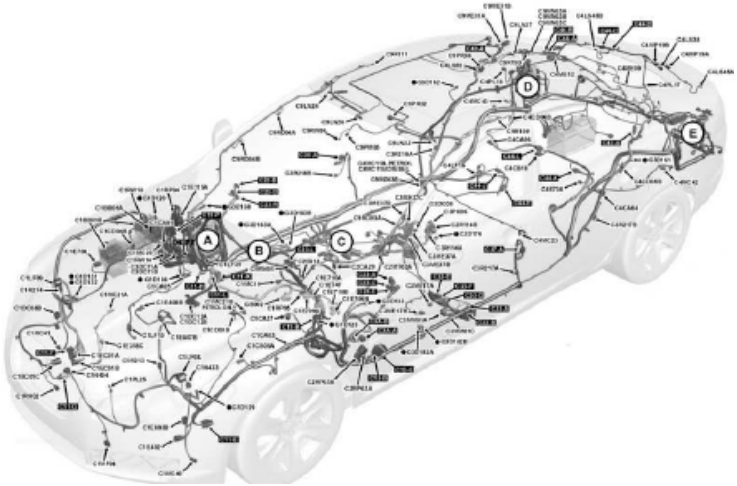

Figure 1.1: Electronic network in a modern road-vehicle (courtesy of Jaguar Cars; source: [Al-Ashaab et al., 2014])

and increases the overall subsystem dependencies and their interconnections.

In the following section, advanced driver assistant systems and remote diagnostics are discussed. These systems are just two of many examples of highly connected, highly complex modern automotive systems. They both contribute to the growing number of interfaces that have been added in the past few years.

1.2.1 Advanced Driver Assistant Systems

Advanced Driver Assistant Systems (ADAS) are systems which assist the driver in driving the vehicle. While early systems could only hold a constant speed and keep the vehicle on track (e.g. adaptive cruise control), modern systems can operate and maneuver the vehicle almost without user intervention.

These ADAS concurrently process data streams from multiple image capturing, distance sensing, and other devices. The high amount of data through-

put requires not only fast ways of data transmission between ECUs and data storage on ECUs, but also a considerable amount of processing power and thus energy.

A less considered requirement of these systems is up-to-date map data and live traffic information. Legally mandatory systems like eCall[1] require a cellular module in all new consumer vehicles sold in the European Union since 2018.

One task of modern ADAS is the recognition of traffic signs and traffic conditions. Even if the location of static traffic signs could be delivered as information added to map data, some signs, and special conditions on a route can change within seconds.

A few examples are:

- construction work on a road
- non-visible road surface markings due to heavy rain
- dangerous traffic conditions caused by a slippery road or pit-holes
- a car crash
- an electronic traffic sign which displays different speed limits depending on the current air-pollution parameters

Many of the above given environmental scenarios are hard to evaluate correctly for computational systems.
A promising solution to this problem is the addition of wireless interfaces to the automotive system which enables them to interact with smart roadside elements like smart traffic signs and other traffic participants in close proximity. This can be used to retrieve valuable information about the environment, which was already gathered by other road-vehicles (for example pot-holes). A promising transmission standard for this task is IEEE 802.11s which utilizes a wireless mesh network structure, see [Chakraborty and Nandi, 2013] and [Steger et al., 2016] for more information. Often used keywords are Vehicle-to-Vehicle (V2V), Vehicle-to-Roadside (V2R), or simply Vehicle-to-X (V2X) interfaces.

[1] see https://eur-lex.europa.eu/legal-content/DE/TXT/PDF/?uri=CELEX:32015R0758

1.2.2 Remote Diagnostics

Diagnostic systems in general provide means to identify problems within the vehicle. If an unknown problem within a vehicle occurs, it is normally transported to a workshop where a diagnostic tester is connected to the standardized diagnostic port of the vehicle. With this diagnostic tester, stored diagnostic trouble codes on the ECUs can be requested and problems can be identified much faster.

A topic which is increasingly attracting the attention of road-vehicle manufacturers is remote diagnostics. In essence, these are services that provide diagnostic functionality over the internet and therefore (depending on the implementation) without physical access to the vehicle, see [Al-Taee, Khader, and Al-Saber, 2007]. These services may even replace current diagnostic hardware interface in future vehicles, see chapter 2.4 for more information.

As with ADAS, remote diagnostics contribute to the growing number of interconnections between the vehicle and the outside world.

1.3 Motivation

As an increasing number of interfaces to the outside world are being added to modern cars, various infotainment and remote-diagnostics services are dependent on internet access. Systems like ADAS often even require a constant data stream from multiple sensors for their safe operation. WLAN, Bluetooth, wireless monitoring systems, e.g. detecting the tire-pressure, and especially their implementation, add additional attack surface.

The electronic architecture in modern road vehicles is the result of several iterations of adding more and more complex components to a closed system, while, at the same time, also complying to the legal requirements of interconnecting them to each other (see 2.4.1).

In the past, cyber-security was hardly ever a requirement in the development of automotive components. The focus was on the proper component

interactions and their compliance with (legal) safety requirements. Diagnostic protocols and their complex identifier compatibility requirements for all ECUs often even led to the weakening of defense mechanisms in favor of ease-of-integration.

This lack of cyber-security awareness paired with adding huge attack surfaces in addition to the very long time-of-life (on average around 15 years, see [Nakamoto, Nishijima, and Kagawa, 2019]), made automotive systems a very vulnerable target.

In addition to this, there is currently no standard specification for cyber-security in automotive systems which addresses the before mentioned challenges. However, in recent years, ISO 26262 (functional safety specification) and SAE J3061 (cybersecurity guidebook for cyber-physical vehicle systems) were used to deal with cyber-security issues: cyber-security was dealt with as a safety issue or a functional security requirement. The standard ISO/SAE 21434 (Road Vehicles - Cybersecurity engineering) (see chapter 2.5.4) is currently available as a draft-international-standard and focuses on cyber-security as a system requirement [Schmittner, 2019].

The problem is getting worse if the potential damage is taken into account. Controlling a fleet of hundreds-thousands of vehicles utilizing the same chain of security flaws for every vehicle and racing them into a city could potentially kill thousands of people. Hijacking an automotive system without having physical access to it has already been demonstrated by [Miller and Valasek, 2015]. Miller and Valasek were able to exploit a flaw in the infotainment system with which allowed them to access the in-vehicle bus-system.
In another case, a security expert by the synonym `L&M` was able to break into thousands of accounts of two GPS tracking services used by automotive manufacturers by reverse engineering their android apps and finding a default password[2]. Through these accounts, he would have been able to remotely turn off the engines of the associated cars.

[2]see https://www.vice.com/en/article/zmpx4x/hacker-monitor-cars-kill-engine-gps-tracking-apps

Since most attacks exploit errors in software, software updates are vital for the secure operation of road vehicles. "*According to IHS Automotive, an auto-industry data consulting company, the cost savings from OTA updates for all the OEMs worldwide are estimated to grow to over $35 billion in 2022.*" [Halder, Ghosal, and Conti, 2019]

1.3.1 Personal Motivation

In 2018 I did a project on automotive keyless entry systems where I implemented a well-known practical attack on a common keyless entry authentication scheme with low-cost components. The attack (RollJam, see [Kamkar, 2015]) was possible due to a rolling code scheme in combination with a transmitting-only key-fob design. As Wouters et al. demonstrated, even modern passive keyless entry systems are vulnerable to attacks, including key-fob cloning, see [Wouters et al., 2019].

Getting access to the interior of a road-vehicle offers a wide array of further attacks and opens it up for thieves. Given the simplicity of the attack and the common appearance in rather new cars, it is particularly hazardous. The only practical solution to such a problem would be the exchange of the key-fob authentication system which also means updating the keyless-entry-system ECU. This entry system vulnerability is a good example of a successful attack due to outdated software. Such vulnerabilities get discovered sooner or later in every software product over its lifetime.

Since almost all new road-vehicles today come equipped with a cellular module, customers can no longer choose to not having a connected vehicle. If there is no remote update system in place and vulnerabilities in a road-vehicle which can be remotely exploited are discovered, customers are basically at the mercy of an attacker. In this situation, without a remote update system, customers can only choose between putting themselves and other traffic participants in danger, or not using their road-vehicle at all until a manual update may eventually be applied in a local workshop.

That is why I decided to make this problem the topic of my master thesis.

2 BACKGROUND AND RELATED WORK

In the following sections an overview of the current (security-relevant) hardware and software components in an automotive system and their interplay is given.

2.1 Automotive System Components

"*On an average, an automotive vehicle today comprises of approximately 100 ECUs and over 100 million lines of software code.*" [Halder, Ghosal, and Conti, 2019] These ECUs are interconnected via different types of automotive bus-systems and are each responsible for a specific task or a set of related tasks. As a comparison, *Boeing's 787 Dreamliner requires about 6.5 million lines of software code to operate its avionics and onboard support systems.*[Charette, 2009] A good visual overview of the lines of code in different technical systems is given by informationisbeautiful.net[1]

The airbag control unit, for example, is responsible for reading three or more acceleration sensors in short intervals, which are strategically distributed near the most common impact zones during a crash. If two or more of these sensors register an unusual high change of acceleration, the airbag control unit has to act. Which airbags have to be activated and at which exact time depends on various factors like which seats are manned, see [Klanner et al., 2004]. Additionally, safety measurements like fuel injection cutoff

[1] https://www.informationisbeautiful.net/visualizations/million-lines-of-code/

and seat-belt pre-tensioning have to be executed. The sensor which registers if a seat is manned may also be used during normal operation by the compartment-interconnection-ECU to alert the infotainment ECU to display a seat-belt warning icon on the dashboard if the sensor in the buckle of the related seat belt registers no inserted clamp.
The above-presented implementation is a classic example of how different component dependencies in a vehicle lead to various system requirements e.g.:

- The airbag ECU has to be hard-realtime capable (the airbag MUST activate within a specified time-frame, otherwise the system is considered to have failed)
- Distributed sensors and ECUs may have an N-N connection
- Means for constant data exchange between different modules across different domains have to be implemented

Wiring each component to every dependent component would be a very complex and very cost-intensive procedure. The solution was bus systems (see section 2.3) which interconnected the components.
Due to the high number of components and increased safety requirement, it should not be possible that non-critical systems like the infotainment system influence the execution flow of the engine control unit in a direct manner, as it happened several times in the past
Therefore, ECUs in modern vehicles are assigned to different domains.
"*Each domain has different requirements. For example, the powertrain domain requires extremely precise timing, closed-loop control, and real-time behaviour, whereas infotainment demands the optimal presentation of information.*" [Mössinger, 2010] ECUs of the same domain are interconnected to each other by an automotive bus-systems to the domain controller. In bus-systems which require a bus-master, the domain controller performs this task. These domain controllers, in turn, are interconnected to each other via an automotive ethernet switch. Fig. 2.1 shows this typical infrastructure of a vehicle containing a wireless vehicle interface (WVI), a diagnostic port controller (DPC), the domain controllers and an automotive ethernet switch connecting them all in a star-topology via automotive ethernet (see 2.3.3). Domain controllers provide an interface for ECUs of the same domain to the rest of the in-vehicle network. In Fig. 2.1, a CAN bus-system (see 2.3.1). connects the domain controller for the body-domain (DC Body) to the ECUs assigned to

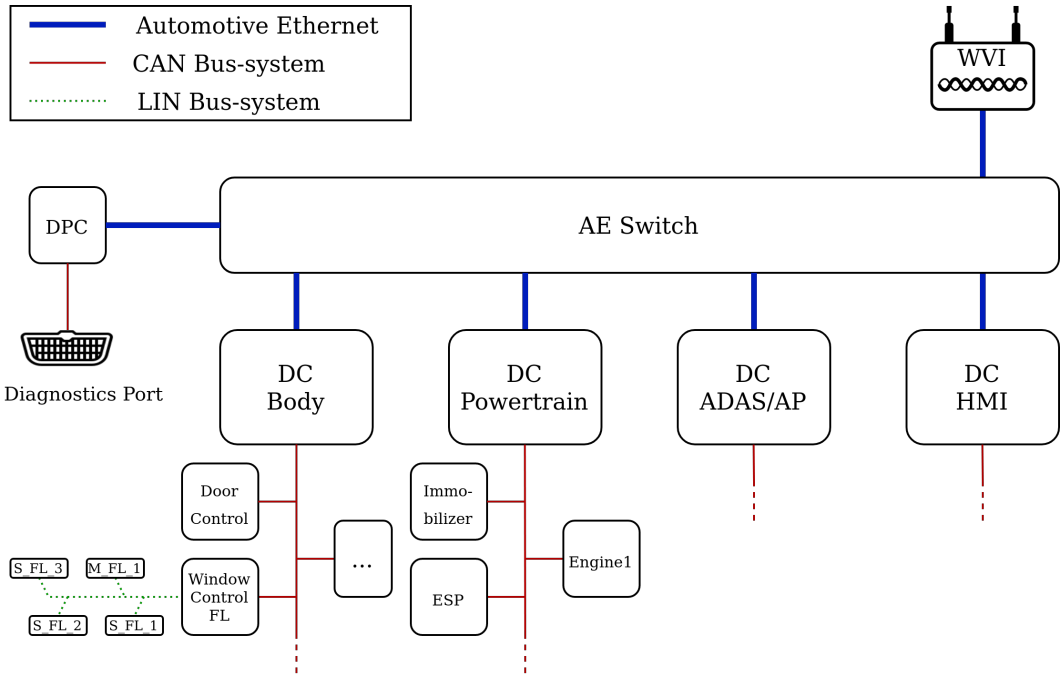

Figure 2.1: Example of a modern In-Vehicle network structure

this domain, like electronic window lifter or other ECUs performing comfort functions. `Window Control FL` in Fig. 2.1 in turn functions as a bus-master for a LIN bus-system (see 2.3.2) connecting sensors and actors related to the front-left window control to it.

2.2 Electronic Control Units

"*An Electronic Control Unit (ECU) in its simplest form consists of a processing unit which is connected to at least one actuator and has a connection to an input signal source.*" [Ribeiro and Baunach, 2017]
The tasks in a car are very complex and depend on an enormous amount of input states which in turn drive an immense amount of output signals. To reduce complexity, tasks of different domains are distributed over many Electronic Control Units (ECU). These control units can be assigned to

Figure 2.2: Tire-Pressure-Monitoring-System ECU

simple tasks like opening and closing a window or very complex ones like fuel injection and related.

What almost all of them have in common is their connection to one or more bus-system and their programmable flash memory (for more information see [Schäuffele and Zurawka, 2003]). In the following chapters, commonly used hardware and software for ECUs are discussed.

2.2.1 Hardware

Almost all ECUs in modern vehicles either use generic micro-controllers or in some cases special automotive micro-controllers. There is a huge variety of performance, architecture, and power consumption.
The development of micro-controllers is very expensive, therefore hardly any automotive company builds its own processors. The development of whole ECUs is outsourced to companies like Bosch, Infineon, and Nvidia. These companies, for example, offer optimized micro-controllers for special requirements or ECU platform designs for specific tasks that can be bought and licensed by automotive manufacturers. An often-used business model is automotive ECU platforms which offer optimization and customization options for vehicle manufacturers (see examples below).
In the smallest variants, ECUs may consist of an 8-bit processor with around

10kB of flash memory which is directly mounted to the circuit board on the driver-seat side window control panel together with four window control levers directly connected to it. Their purpose may be to react to a hand full of predefined LIN message with a predefined response and sending out a hand full of prefixed LIN messages depending on the state of the window switches. More about LIN-Messages can be read in section 2.3.2.
On the other scale of the spectrum, there are for example the Intel Atom A3900 processor series[2] with a dedicated graphics processing unit which may drive the main touch display on the dashboard or the Nvidia Jetson modules being able to process gigabytes of data from several image capturing and other high-resolution sensors simultaneously. With these multicore-processors, new opportunities and challenges for arise, see [Gai and Violante, 2016] for more information. Other examples of dedicated purpose processing solutions for the automotive domain are:

- XA Spartan FPGA product line (by Xilinx)
- R-Car SoC product line (by Renesas)
- Jacinto™ TDAx ADAS-SoCs (by Texas Instruments)
- S32 Automotive Processing Platform (by NXP)
- 32-bit AURIX™ Microcontroller (by Infineon)

A very popular example is the AURIX™ TriCore™ product line from Infineon. These processors use three cores which are all assigned the same task and hence perform the same operations. Their outputs are compared to detect any discrepancies between the results. If however, an error occurred (e.g. caused by a bit flip) in one core, the correct result can still be determined by a majority vote on the processing results of all cores. Since the same simultaneous error in two cores is highly unlikely, this processor type is often used for safety-critical functions like the airbag-control-unit.

2.2.2 Gateway ECUs

A special type of ECU are so called gateway ECUs, which are typically used for two main applications: diagnostics and network gateways.

[2] see https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/atom-processor-e3900-a3900-series-datasheet-addendum.pdf

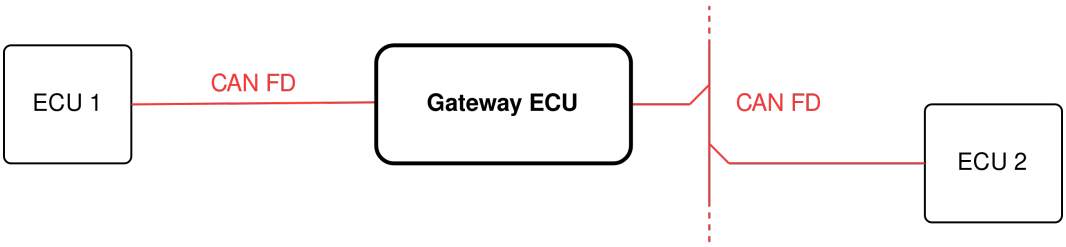

Figure 2.3: Diagnostic Gateway ECU

2.2.2.1 Diagnostic Gateway ECUs

As described in 2.2.1, many different ECUs from different manufacturers are built into modern automotive systems. The problem is that these ECUs may use different or already existing manufacturer specific Diagnostic IDs (DID) for the same diagnostic message. Their diagnostic functionality in this case is simply incompatible (see 2.4 for more information). The easiest way for vehicle manufacturers is often to mask the data traffic of an otherwise manufacturer-diagnostic-incompatible ECU with a diagnostic gateway. Its purpose is to adjust incoming data (exchange the DID, alter data formats, etc) in order to make non-standard diagnostic messages compatible with the desired diagnostic specifications of the automotive system. Examples are given in Fig. 2.3 and in the Patent Documents by Susumu Akiyama, see [Susumu Akiyama, 2015].

2.2.2.2 Network Gateway ECUs

To connect different bus-systems and make data exchange possible, a gateway is needed. Its purpose is to store and forward messages from one bus-system to one or more other bus-systems. These bus-systems can be of the same or different kind. In the latter case, the gateway is also assigned to the task of translating data formats and is responsible for message routing. Its correct implementation is essential for the security of the automotive system. For example, a direct shut-down command from the BCM (Body Control Module) to the ECM (Engine Control Module) should be discarded. Therefore, most gateway ECUs use processors with higher clock speeds

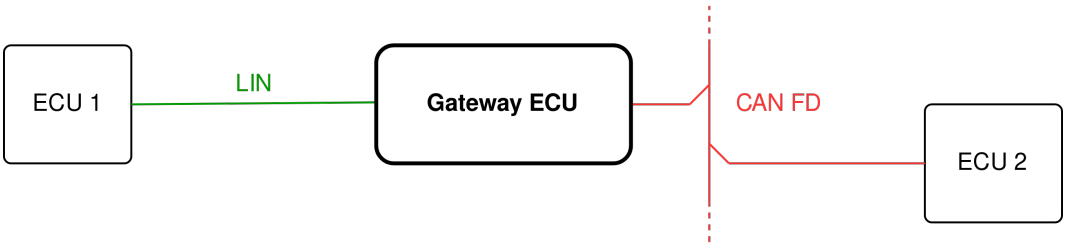

Figure 2.4: Bus Gateway ECU between a LIN and a CAN network

and contain more flash memory than normal ECUs. Depending on the transmission speeds of the connected bus systems, these processor clock speeds can be up to a few Gigahertz.
In 2.4 a bus gateway configuration is shown. This configuration is used if the desired component does not support the required bus interface.

2.2.3 Software

The software in automotive systems is bound to various legal requirements in terms of safety and certain functionalities in many countries. These requirements are often derived from norms and standards published by organizations like the International Standard Organisation. For more information on norms and standards and their impact on automotive systems, see section 2.5.

An automotive system consists of many overarching logical functions stretching across ECU boundaries which implement these legal regulations and every other OEM-defined functional requirement. In order to implement these functions in a cost-effective way, automotive software specifications like OSEK-OS 2.5.7, AUTOSAR (which is based on OSEK-OS), and JasPar were defined.

2.2.3.1 Flashing And Coding

Flashing and Coding describe the process of how car manufacturers reuse the same base component for many variants of vehicles, in order to save money.

As mentioned previously, an automotive system is composed of many different components with different (overarching) functionality. In order to cover a wide range of user preferences and purchasing powers, road-vehicle manufacturers often develop different variants of the same car. These variants differ in features and functionality. For example, the adaptive-drive-assistant-system (ADAS) of the base model of a car may only be equipped with cruise control while the mid-range model would also feature lane guidance and autonomous driving to some degree.

Sometimes a feature is just not allowed in some countries. In order to map these functionality differences into software, without rewriting the whole codebase of a road-vehicle, Flashing and Coding is used.

Every functionality which an ECU may hold is written on to that device via the Flashing process. This functionality is embedded in the process layer of an OSEK-OS conform ECU operating system via one or multiple processes (see section 2.5.7 for more information).

Flashing is done before the ECU enters the assembly line. At the end of the assembly line, when all electrical parts of the vehicle are combined and connected, the Coding process takes place. This is usually done by connecting a manufacturer-specific diagnostic device to the diagnostic port of the vehicle. See 2.4 for more details. This device uses manufacturer-specific diagnostics commands to message every ECU which processes should be enabled and which should be disabled. This information is stored in the so-called Coding Sequence. It holds information about all enabled processes on an ECU and acts as a table of processes that can be scheduled. There are also overarching functions which consist of several processes on different ECUs. In order to keep overarching consistency (either activate or deactivate all interrelated processes of the same functionality), so-called Coding Strings are used.

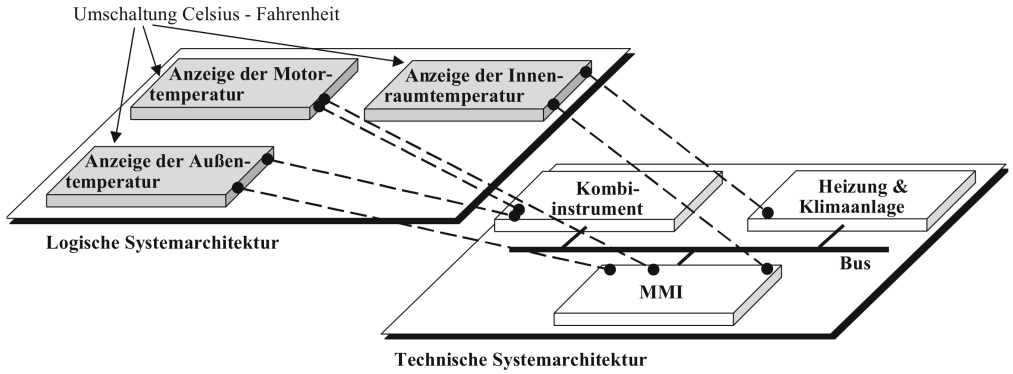

Figure 2.5: Overarching functionality in an automotive system [Schäuffele and Zurawka, 2003]

These consist of a few bytes of data and describe the Vehicle Operations and the type of vehicle variant (base model/pro model/..). Coding Strings are often stored on two or three different hard to replace ECUs. Prominent ECUs to store the Coding String are the Body-Control-Module (BCM) and the Instrument-Panel-Cluster (IPC) (an example is given in [Mercedes-Benz Canada, 2005]). Coding Sequences (per ECU) and the Coding String (per vehicle) define the entirety of functions in a vehicle. They are often the only difference between two variants of a road-vehicle. It is sometimes just more cost-efficient to build one type of electrical architecture with one code base and then generate two-vehicle variants out of it at the production line by connecting a manufacturer-specific diagnostic device and running the coding process.

2.3 Bus systems

ECUs in an automotive system are connected via various bus systems. In the following, the most popular bus system types are discussed: LIN, CAN, CAN-FD, and Automotive Ethernet.

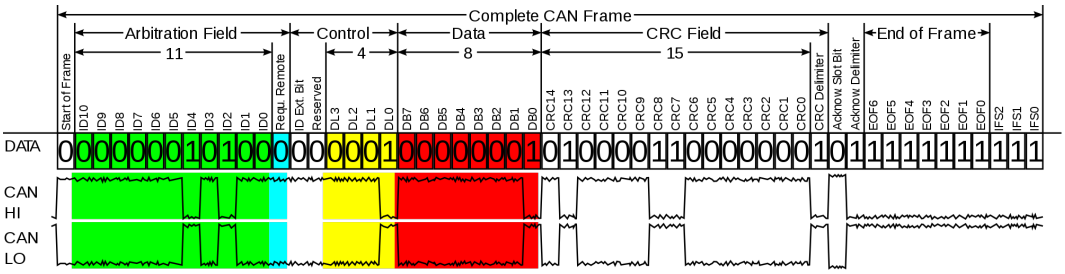

Figure 2.6: CAN base frame structure (licensed under the Creative Commons Attribution-Share Alike 3.0 Unported by Erniotti)

2.3.1 CAN Bus

The Controller-Area-Network (CAN) is a multi-master serial bus system. It is used to connect multiple ECUs of the same functional domain. The CAN bus-system is standardized in ISO-11898 series for road-vehicles, see [International Organization for Standardization, 2003] for more information about the physical layer. The maximum baud rate for High-Speed CAN with normal frame size is 1Mbit/s. Physically, CAN is based on two bus wires: CAN Low and CAN High. In the recessive state, both bus wires are at $2.5V$. In the Dominant state, CAN High is at $3.5V$ and CAN Low is at $1.5V$. If multiple nodes apply different states at the same time, the Dominant state always overwrites the Recessive state.

The **SOF** (Start of Frame) consists of one dominant bit which indicates the start of a new frame.

The **Arbitration Field** consists of 11 ID-bits and one RTR (remote-transmission-request) bit. There is no bus master in CAN. Every bus node which wants to transmit a message transmits its identifier-bits while comparing them to the actual bus state. If one ECU notices a difference between the bus-state and its transmitted state, it stops its transmission until the next frame. This happens when the node transmitted a recessive bit but another node transmitted a dominant bit. Therefore, the message with the most dominant bits at the beginning of the identifier wins the bus access and can transmit the rest of its message without interference. The arbitration field indicates the message priority. The more dominant bits an identifier starts

with, the higher the message priority is. The RTR bit indicates whether the frame contains data (dominant/0) or the sending node requests data from the receiving node (recessive/1). If Bit 12 (RTR-bit in base frame) and Bit 13 (IDE in base frame) from the arbitration field are recessive, the frame is an extended CAN frame. This extended CAN frame has a longer arbitration field which consists of an additional SSR bit, an IDE bit, and 18 extended ID-bits between the 11-bit base ID and the RTR bit. Since the SSR and IDE bit in the extended frame are the Bits 12 and 13, both are recessive.

The **Control Field** consists of the IDE bit, the $r0$ bit, and four DLC bits. The IDE bit indicated whenever the containing frame is a base frame (dominant) or an extended frame (recessive). The $r0$ bit is reserved for future use and therefore always dominant. The four DLC (data-length-code) bits contain information about the number of data-bytes in this frame or the requested frame.

The **Data Field** contains the payload data (0-8 bytes).

The **CRC Field** consists of 15 CRC bits and one CRC delimiter. The CRC bits are the result of the CRC computation with all previous bits of the CAN frame. The CRC delimiter indicates the end of the CRC field and is always recessive.

The **ACK Field** consists of an acknowledge-slot-bit and an acknowledge-delimiter-bit. The transmitting node sets the acknowledge-slot-bit in a recessive state. Every node which has received an error-free frame so far has to set the bus in the dominant state during this bit transmission. By monitoring the bus state, the transmitting node can verify if the message has been received by at least one other node.

The **EOF** (End-Of-Frame) consists of seven recessive bits. If a dominant level occurs, the frame is invalid.

Three or more **IFS** (Inter-Frame-Spacing) bits, all recessive level, are between the EOF of one frame and the SOF of the next one.

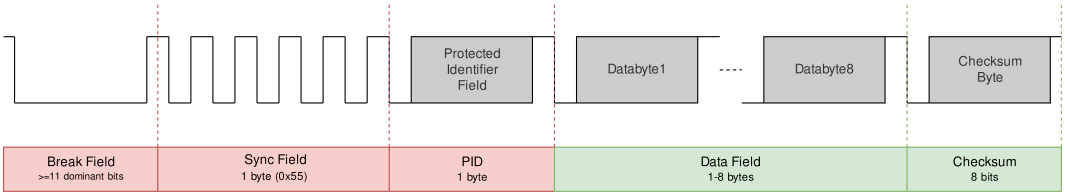

Figure 2.7: LIN frame structure

2.3.1.1 CAN-FD

The classic CAN frame contains a lot of overhead and can only contain 8 bytes of payload data. Therefore CAN-FD (Flexible Data-rate) was developed. The key differences are:

- Flexible data rate: the maximum bit rate during the arbitration phase remains at 1Mbit/s, but can be set up to 5Mbit/s during the data phase.
- Up to 64bytes of payload data
- The CAN-FD Frame contains a dedicated Stuffing-Bit-Count field consisting of four bits in order to count the number of inserted stuffing bits
- The CRC field has been extended to 17 bits for 0-16 data bytes and 21 bits for 17-64 data bytes. The Stuffing-Bit-Count field is also included in the CRC computation.

For more information on CAN-FD see [Bosch, 2012].

2.3.2 LIN Bus

The Local-Interconnect-Network (LIN) consists of three physical connections:

- Vcc
- Data
- Ground

The constant voltage levels on Vcc and Ground are derived from the battery voltage levels. The voltage on Data determines the bus level. A voltage level on Data between 100-80% of the battery voltage means a Recessive state, a voltage level between 0-20% of the battery voltage means Dominant state. On concurrent bus access, the Dominant state overwrites the Recessive state.

LIN is intended as a standardized protocol for the communication of an ECU and multiple related sensors and actuators. Its maximum baud rate is 20kbits/s and one Frame can contain 1-8 bytes of payload data. A LIN network is composed of one Bus-Master and up to fifteen slaves.

The Master sends out a Frame Headers to which the addressed Slave must provide a response. The LIN frame format is shown in 2.7. The Sync Byte, the PID, the bytes of the Data field, and the Checksum Byte are all transmitted as UART frames. Break Field, Sync Field, and PID are transmitted by the Master (marked red in 2.7). Data Field and Checksum are transmitted by the addressed Slave (marked green in 2.7).

- The **Break Field** consists of at least 11 dominant bits. It is used to signal the Slaves that a new frame starts.
- The **Sync Field** is consists of an alternating sequence of recessive and dominant states and is used by the Slaves to detect the Master Baud Rate.
- The **PID** contains a six-bit frame identifier and two parity bits. Each network participant holds a LIN Description File (LDF) which defines all possible PID values. Using this file the response type, the Data Field length and the addressed Slave are defined.
- The **Data Field** contains the 1-8 bytes of data sent by the slave.
- The **Checksum** is calculated by the slave and contains either a Checksum based on all data bytes or a checksum based on all data bytes and the PID.

2.3.3 Automotive Ethernet

When talking about automotive ethernet, one refers to two standards:

- BroadR-Reach
- IEEE 100Base-T1/1000Base-T1

The main difference between automotive ethernet and normal IEEE-802.3 ethernet is the definition of the physical layer. In the following, both automotive standards are discussed

BroadR-Reach is an open standard for a physical layer of a point-to-point Ethernet connection for the automotive domain. It was developed by Broadcom® and standardized by the OPEN Alliance SIG as the OPEN Alliance BroadR-Reach PHY. This physical layer consists of one single pair of unshielded twisted pair (UTP) cable. *BroadR-Reach Physical Layer (BR-PHY) Transceiver supports standard media access controller (MAC) interfaces and can therefore be combined with IEEE Standard 802.3 compliant switch technology* [Broadcom, 2014]
It provides a data rate of 100Mb/s, is specifically designed for the requirements of the automotive industry, and combines features of IEEE Gigabit 1000Base-T and IEEE 100TX. Furthermore, its bandwidth was reduced to 33.3Mhz compared to 1000Base-T and 100TX, which enables it to comply with automotive regulations regarding electromagnetic emission. Since it utilizes full-duplex transmission, it also supports echo cancellation. For more detail on BroadR-Reach see [Broadcom, 2014].

"IEEE Reduced Twisted Pair Gigabit Ethernet" is the IEEE adoption of BroadR-Reach. Both standards are nearly identical, are interoperable and both are often used interchangeably with the term automotive ethernet. The minor differences are for example at the timing specification on the wake-up commands. The IEEE standard also considers other fields of application like in aviation.

The advantages of automotive ethernet are:

- low cost due to single twisted cable pair
- low cable weight
- high data rates
- supports MAC interfaces
- supports higher ethernet protocols

For hard real-time network traffic, the time-triggered ethernet extension (TTEthernet) exists. Hank et al. noted that "*...the Time-triggered ethernet extension (TTEthernet) allows standard best-effort communication and hard real-time network traffic to share the same layer 2 infrastructure*" [Hank et al., 2013].

2.4 Diagnostics

Diagnostic capabilities are a rather old functionality in road-vehicles, dating back to the 1970s. Diagnostic Capabilities in a road-vehicles split into two major groups:

- On-Board-Diagnostics
- Off-Board-Diagnostics

In the following, a short introduction to On- and Off-Board-Diagnostics is given.

2.4.1 On-Board-Diagnostic

On-Board-Diagnostics (OBD) is the ability of a vehicle to identify, store, and report errors and monitor special system procedures. Since a road-vehicle is a complex system, the source of errors and malfunctions are often hard to determine. Therefore most ECUs constantly perform diagnostic tasks during the operation of the road-vehicle and some continue even after the ignition is turned off.

These tasks could be as simple as periodically requesting a value from a sensor and testing it against an out-of-range condition or more complex like looking for specific system behaviors by applying advanced calculus onto multiple system value array.

If abnormal behavior is detected, the responsible ECU generates and stores a corresponding Diagnostic-Trouble-Code (DTC). DTCs are predefined for each ECU in each vehicle in so-called "Summary Tables" and are also important for the registration of new vehicle types by the manufacturer.

Repeatedly stored DTCs can lead to the Malfunction-Indicator-Light (MIL) on the Dashboard to turn on. It is a simple tool to inform the driver that there exists a problem with the vehicle and a visit to the workshop may be necessary. In some cases, these DTCs can even prevent a vehicle from starting up for safety reasons. Besides "Generic DTCs", which are defined in ISO 15031-6 and SAE J2012, there are also manufacturer specific DTCs (see 2.4.2.1).

2.4.1.1 Monitoring Emission relevant components

The monitoring of emission-relevant components holds a special place in On-Board-Diagnostics since these self-diagnostic capabilities and the resulting DTCs are mandatory in almost all major car-markets like the US, the EU, China, and Japan. Monitoring requirements according to UNECE Regulation 83.07 Annex 11 [Economic Commission for Europe of the United Nations, 2019] range from Engine-Misfire occurrences over Evaporative-Emission-Purge-Control (EVAP), up to Tank-Leakage detections and NOx after-treatment devices. Monitoring areas depend on the propellant system of the road-vehicle, see [Farrugia et al., 2017]. NOx after-treatment device monitoring, for example, is not mandatory for gasoline-powered road-vehicles, EVAP system monitoring is not required for Diesel-powered road-vehicles and BEVs do not contain any emission relevant components at all.

2.4.2 Off-Board-Diagnostic

Off-Board-Diagnostics describes the diagnostic functionality outside the physical boundaries of the road-vehicle. DTCs indicate the occurrence of specific undesired system behaviors. The interpretation of these DTCs is done by a diagnostic tester. A Diagnostic tester usually consists either of a small electronic device with a connection cable to the OBD-II port of the road-vehicle, or of an adapter from the OBD-II of the road-vehicle to a standard computer interface with client software running on this computer. A novel approach is given by Yang et al., see [Yang et al., 2013]. The OBD port is a standardized port which every road-vehicle produced after 1996

has equipped and is connected to the internal bus system of the road-vehicle.

If a vehicle encounters some kind of malfunction or error (e.g. an ignition plug is broken), the mechanic at the workshop plugs in the diagnostic tester to the OBD port and requests the stored DTC codes from the ECU memories. Most diagnostic testers offer a so-called "guided diagnostics" which utilizes the DTCs in combination with fault trees to track down the exact source of the error, see [Satnam Singh, Bangalore (IN); Vineet R. Khare, Bangalore (IN); Rahul Chougule, 2013]. This makes the identification of errors much easier and decreases maintenance costs for the customer and the workshop. Diagnostic testers are also able to erase DTC codes from the ECU's memory e.g after the problem which caused the DTC is fixed. This can also be performed by the ECU itself for some faults if they have not been detected for a specific time.

2.4.2.1 Manufacturer specific Diagnostics

In addition to the mandatory emission-relevant DTCs and the normed but optional generic non-emission relevant DTCs, there are also the manufacturer-specific DTCs and additional manufacturer specific diagnostic capabilities. Manufacturer-specific diagnostic functionality is often not publicly accessible and contains additional system monitoring capabilities. This gives the manufacturer workshops an advantage over a generic workshop in terms of troubleshooting.

2.5 Norms And Standards

In the automotive industry, norms and standards have a special significance. Countries derive their regulations and legal requirements, e.g. for automotive safety requirements or on-board diagnostics, for road-vehicles from these standards. Therefore, automotive manufacturers are often able to comply with different legal requirements from different countries by complying with international norms and standards. This is especially true for automotive safety norms and standards. Safety engineering today affects

every mechanical and electrical part of a modern vehicle and is a huge cost-factor during development.

This section gives an overview of automotive standards which are the most relevant for this work:

- IEC 61508 ”Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems”
- ISO 26262 ”Road vehicles - Functional safety”
- SAE J3061 ”Cybersecurity Guidebook for Cyber-Physical Vehicle Systems”
- ISO/SAE 21434 ”Road vehicles - Cybersecurity engineering”
- UNECE WP.29 (Activities on Automotive Cybersecurity and OTA)
- ISO/AWI 24089 Road vehicle - Software update Engineering
- ISO 17356 Open interface for embedded automotive applications

The reason safety standards are also discussed here is that safety and security in the automotive domain depend on each other and as Schmittner et al. described it:“*They both focus on system-wide features and could greatly benefit from one another if adequate interactions between their processes are defined.*” [Schmittner and Macher, 2019]
“*A concept is gaining momentum that security and safety are closely interconnected. Nowadays it is not acceptable to assume that a cyber-physical system is immune to threats and it is not feasible to assure the safety of the cyber-physical system independent of security.*” [Chowdhury et al., 2018]
Security-related standards for the automotive domain from various organizations are currently in development or have been released in the last few years.

2.5.1 IEC 61508

IEC 61508 “Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems” focuses on E/E/PE systems and their components those failures (alone or combined with other failures) could lead to injury or death of humans or damage to the environment and/or the product itself. It functions as a general norm for all safety-related systems across the industry.

Hazard and risk analysis consider ”likelihood” and ”consequences” categories of failures which can be used to deduce the associated safety integrity level (SIL). Threat analysis itself or dedicated procedures are not specified. IEC 61508 is the foundation for ISO 26262.

2.5.2 ISO 26262

The ISO 26262 ”Road vehicles - Functional safety” is based on IEC 61508 and therefore also based on risk assessment. It not only focuses on the whole development process and life-cycle of road-vehicles but also provide risk-level-classes, specifically tailored for the automotive industry called ”Automotive Safety Integrity Level” (ASIL). The risk of failure of safety-related systems, electronic and electrical systems as well as related mechanical subsystems to some degree, is evaluated and countermeasures are defined to achieve the desired ASIL level.

Unlike IEC 61508, guidelines for the interaction between security and safety are defined (especially Annex E). ISO 26262 was defined with the upcoming ISO/SAE 21434 standard in mind (see 2.5.4).

2.5.3 SAE J3061

SAE J3061 ”Cybersecurity Guidebook for Cyber-Physical Vehicle Systems” is a predecessor of ISO-SAE 21434 and establishes a set of high-level guiding principles for cybersecurity. [Schmittner and Macher, 2019]
It describes a complete lifecycle process framework which is similar to the safety-related framework in ISO 26262 2.5.2. A threat-assessment and a risk-assessment for all cyber-security systems are proposed. Restrictions on interfaces between safety and security management and processes and their interaction are not defined. However, it is acknowledged that cyber-security-related systems in case of failure may affect safety-related systems and therefore the required safety levels.

The application of SAE J3061 is described in [Skavhaug et al., 2016] in detail. SAE J3061 is no longer available and is now under rework to distinguish it from ISO-SAE 21434.

2.5.4 ISO/SAE 21434

The ISO/SAE 21434 "Road vehicles - Cybersecurity engineering" is set to be a corner-stone for cyber-security related systems in automotive systems. It is currently under co-developed by working groups of both ISO and SAE.

The standard, although designed for road-vehicles, will not include specific countermeasures, implementation suggestions, recommended cryptographic algorithms, or cover unique requirements for autonomous vehicles. The standard will propose a risk factor analysis for prioritization of measurements on cyber-security relevant systems. *Additionally this standard will considers cybersecurity activities/processes for all phases of vehicle lifecycle, ranging from design and engineering, production, operation by customer, maintenance and service and decommissioning.* [Barber, 2018]

ISO/SAE 21434 is currently available as a draft-international-standard. The final standard is set to be released in November 2020.

2.5.5 UNECE WP.29 (Activities on Automotive Cybersecurity and OTA)

Instead of different requirements on vehicles and their components in every state, United Nations Economic Commission for Europe (UNECE) aims to provide a general standard for over 60 contract states in all relevant topics of type approval. This not only facilitates type approval and trading but also benefits OEMs which do not have to re-develop entire parts of automotive systems for each member state. Note that these regulations only affect components and systems relevant for type approval. UNECE set up a working party (WP.29) in order to harmonize national regulations on vehicle type approval for cyber-security and OTA capabilities. In this chapter, only the activities on OTA are discussed.

The structure of the final report[3] on OTA contains recommendations on software update procedures. These recommendations are split into the main body, Annex A and Annex B.

The main body (Software update guidance) consists of six chapters and gives addresses general processes and procedures for software updates. Furthermore, OEMs are advised to support national registration processes. For more information on Annex A and B, see CS/OTA Task Force [4].

2.5.6 ISO/AWI 24089 Road vehicle - Software update Engineering

This standard currently only exists as a Working Draft and is not publicly available. A committee draft is set to be finished at the end of March 2020 and a DIS (draft international standard) publication is set to be released in April 2021. The final publication is set for March 2022.

Schmittner et al. describe it as follows: *'It defines requirements like ensuring that the vehicle is in a state where an update is feasible, restrict vehicle functionality during the update process, test the updated system in the vehicle, recover from failed or interrupted updates and re-activate vehicle functionality and operation'* [Schmittner, 2019]

The focus on the second draft, which is currently under development, is set on the software update management system and the requirements for download, install, and activation.

[3]see Final Report of TF-CS/OTA:
https://www.unece.org/fileadmin/DAM/trans/doc /2018/wp29grva/GRVA-01-19.pdf

[4]CS/OTA Task Force (Meeting Documents):
https://wiki.unece.org/pages/viewpage.action?pageId=40829521

2.5.7 ISO 17356 Open interface for embedded automotive applications

ISO 17356 is concerned with OSEK/VDX Operating System. This standard focuses on OSEK-OS and specifies interfaces between the OS itself and its application software. It was developed as a standard real-time operating system environment for automotive control units in which tasks (application processes in the OSEK-OS) can be executed according to a scheduling algorithm.

Tasks and other objects like data structures or events are statically defined before compilation in OSEK implementation language (OIL, defined in Part 6 of the standard) files and can, therefore, be easily ported to other ECUs. Tasks split up into two different types: Simple Tasks and Extended Tasks.

While Simple Tasks can be in the following states: [Running, Ready, Suspended], extended tasks additional have a state called [Waiting] which enables them to wait e.g. for the acquisition of a Mutex to access a resource. Tasks have predefined priorities and can be interrupted by tasks of higher priority if pre-emptive scheduling is used.

AUTOSAR OS[5], a popular OS specification in the automotive industry, is based upon ISO 17356-3 and is therefore backward compatible to OSEK-OS.

For more information about OSEK-OS and its components, see [OSEK/VDX, 2005].

2.6 Existing Update Solutions

In the following sections, two existing update systems are presented: UPTANE and ASSURED. UPTANE is considered the de-facto standard for

[5]see https://www.autosar.org/

updates on automobiles due to its compromise-resilient design. ASSURED improves upon UPTANE by adding remote attestation. Both systems were chosen since their The-Update-Framework-based design is similar to the proposed BECAUSE framework and therefore more comparable.

2.6.1 UPTANE

UPTANE (see [Kuppusamy et al., 2016]) is based on a modified version of The-Update-Framework (TUF)[6]. Since TUF has already been described in many different scientific publications, the following summary consists mostly of referenced excerpts:

The-Update-Framework was originally designed to secure the distribution of data by software repositories. The tasks of the software repository are divided into four roles which should be (physically) separated from each other. Each role is responsible for signing different parts of the payload metadata. The goal is not to prevent attacks at all, but to devitalize them before any changes to the client systems are made. This ability is called `compromise-resilience` and is based on the divide-and-conquer principle. TUF additionally offers means to recover from attacks by revoking the keys of compromised roles.
The following basic roles in TUF are defined:

The **Timestamp** role metadata contains a list of sizes and hashes of snapshot-metadata and therefore indicates if there are any new images and metadata on the update system.

The **Snapshot** role metadata holds a list of Director-, Target- and Root-Metadata and their versions and therefore indicates which files belong to the current release.

The **Target** role metadata consists of hashes and sizes of the actual data like images the clients want to obtain. The Target role may delegate the responsibility of signing metadata about images to multiple custom-made

[6] see https://theupdateframework.io/

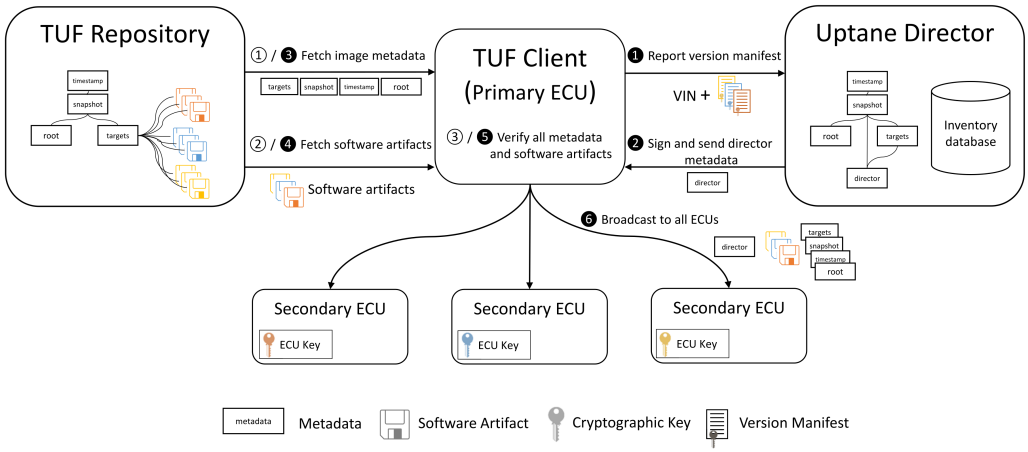

Figure 2.8: Sequence of update distribution events in TUF and UPTANE (source: [Asokan et al., 2018])

sub-target roles to split up this task based on various factors. An example would be that a software image might be provided by different maintainers where the maintainers sign their images. A delegation binds the public keys used by a delegatee to a subset of the images these keys are trusted to sign. [Kuppusamy et al., 2016]

The **Root** role serves as the certificate authority of TUF. Its job is to distribute and revoke the public keys used to verify metadata produced by each of the four roles (including itself). [Kuppusamy et al., 2016]

The **Director** role was added to customize TUF for automotive systems. It functions as a remote package manager for automotive systems and is tasked with image dependency resolution for the ECUs. As mentioned in 2.2.3.1, different vehicles of the same type may have different functions enabled and therefore require different software images and image configurations. The Director's job is to assemble a signed list of update instructions each ECU must perform based on the received Vehicle-Version-Manifest.

2.6.1.1 Update Distribution Process

The update sequence of TUF is shown in Fig. 2.8 and consists of ①-③. First metadata of all roles fetched by the TUF-Client ①. According to them, the required software artifacts are fetched ②. Finally, the TUF-Client verifies and installs the software artifacts ③.

Fig. 2.8 also depicts the update distribution process as proposed by UPTANE. One ECU in the automotive system is appointed as "Primary ECU" the rest are referred to as secondary ECUs. The primary ECU requests signed information about the current software version (referred to as the version manifest) of each secondary ECU. The primary ECU adds the VIN of the road-vehicle and sends this data to the UPTANE Director ①. The director performs a software dependency resolution and determines the new software configuration for each ECU. This director metadata is signed and sent back to the primary ②. The primary ECU then fetches the required image metadata and determines the software artifacts (SA) to be fetched ③. The primary then downloads and all the required software artifacts ④. These software artifacts are checked against their corresponding metadata by the primary ECU ⑤. Metadata and software artifacts are then broadcasted by the primary ECU to all secondary ECUs over the local bus-systems ⑥.

Since not every ECU has the computational power and memory capacity to verify these hashes and store all metadata (full verification), there is a second method called partial verification. In this method, the secondary ECU only verifies and stores director metadata.

2.6.1.2 Features

In the following paragraphs, some features which are supported by UPTANE are listed. These features describe typical attack scenarios and how they can be prevented.

Prevent endless data attacks
ECUs can utilize additional storage to keep the previous image or at least a

delta file, highlighting the changes between the current and the previous image. This not only prevents endless data attacks but also allows the bootloader of the ECU to fall back to the previous image if the update process failed.

Prevent mixed-bundles attacks
The primary ECU broadcasts director- and other metadata to all secondary ECUs. Since all secondaries receive the same metadata, mixed-bundle attacks are prevented. *Ideally, this is implemented by the in-vehicle network (e.g., Ethernet broadcast or CAN bus) such that any metadata sent to one secondary by a primary will be seen by all other secondaries at the same time.* [Kuppusamy et al., 2016]

Prevent partial bundle installation attacks
ECUs sign their installed version number with their public key and send this information to the primary which in turn sends this information to the director. Partial bundle installation attacks can not be prevented but using the vehicle-version-manifest, these attacks can be detected by the director if it is not compromised.

Prevent freeze attacks
To prevent freeze attacks due to inaccurate clocks on ECUs (including primary ECU), all ECUs have to update their clock with an external time server at regular intervals. *Given this as a precondition, freeze attacks are limited to the earliest expiration of timestamp of the root, the director, or a targets metadata file.* [Kuppusamy et al., 2016]

2.6.2 ASSURED

Although designed for large IoT device clusters, the ASSURED framework (see Asokan et al., 2018) can also be applied as an automotive firmware updates system. ASSURED utilizes TUF as a middleman between the OEM and the client system.

Instead of adding an additional role to the TUF, ASSURED tasks the OEM to generate one authorization tokens per software artifact. This allows the OEM,

much like the director in UPTANE, to apply individual constraints on each software artifact. As an alternative, the GlobalPlatform TEE Management Framework [Technology, 2016] can be used to enclose the authorization token in an updated envelope, which is then delivered to the client device. These two approaches can both be realized in ARM Trust-Zone-M and HYDRA architectures [Asokan et al., 2018].

Unlike UPTANE, ASSURED includes the client devices in their security concept while keeping the computational burden on the update-receiving devices relatively low (see prototype implementations at [Asokan et al., 2018]).

What is particularly interesting about this framework is its end-to-end security despite the resource constraint of IoT devices and the transfer of computationally intensive tasks to other devices in the update chain.

Its main disadvantage is that it is not specifically designed for automotive systems and its stakeholders. In the following section, the update process is described in detail.

2.6.2.1 Update Distribution Process

Here the update distribution process, as illustrated in Fig. 2.9, is described in detail.

The OEM creates software artifacts (SA) **2.9** ①. For each SA an authorization token is computed which includes constraints on the target device, metadata of the SA, the hash of the SA, and a signature computed with the OEM authorization key and the content of the authorization token. The SA and its corresponding authorization data are bundled to form the update envelope.

The authorization token can either be placed in the update envelop using the TEE Management Framework (TMF) or directly in the TUF target metadata. For more information see [Technology, 2016].

The update envelope together with its metadata is uploaded to the ”Software Distributor” which resembles the TUF repository ②. The ”Domain Controller” in ASSURED has a similar role as the ”Primary ECU” in UPTANE. It

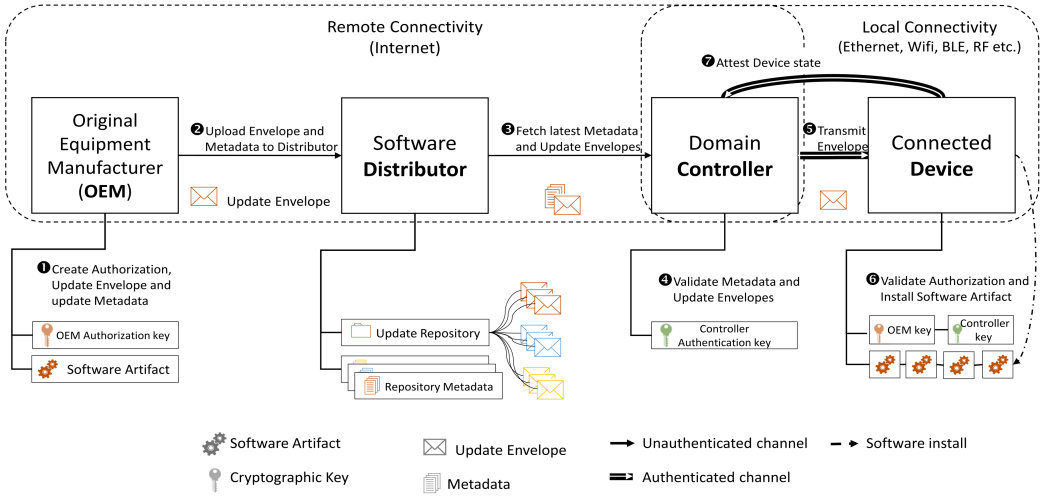

Figure 2.9: Sequence of update distribution events in ASSURED (source: [Asokan et al., 2018])

fetches the latest target and snapshot metadata, fetches the required update envelopes intended for "Connected Device" according to the metadata ③. The envelope(s) are then validated through the target metadata ④.

In **2.9** ⑤ the update envelope is transmitted to "Connected Device" via an authenticated channel. "*This channel serves as an implicit authorization from controller that it has approved the SA in the transmitted update envelope. The Device uses its underlying security architecture to securely validate authenticity and integrity of the OEM's authorization token and SA in the update envelope.*"[Asokan et al., 2018]

If the validation is done and the constraints for the intended target device are meet, "Connected Device" applies the software changes ⑥. After the software changes have been applied the "Connected Device" attests to the "Domain Controller" that it has actually applied the software changes ⑦. *The last step allows controller to obtain a verifiable proof when the update process is complete. Meanwhile, if the update process fails (e.g., by the adversarial preventing an update from reaching device), the Controller will be able to detect it due to the incorrect or missing response.*[Asokan et al., 2018]

3 PROBLEM

Today, various interfaces connect automotive systems to the outside world and the amount of ECUs and contained software increased enormously. "*An automotive system includes a very diverse range of components with a total of more than 100 million lines of code*" [Halder, Ghosal, and Conti, 2019]. Vulnerabilities in software are therefore not only likely to be discovered but also much easier to be remotely exploited. For example, remote code execution-exploits pose a security violation to the vehicle which influences the vehicle functionality and can therefore compromise the safety of vehicle passengers, pedestrians, and other traffic participants. Hijacking an automotive system without having physical access to it has already been demonstrated by [Miller and Valasek, 2015].

The only countermeasure to this problem is to provide means for updating the in-vehicle software if a vulnerability is reported.
An update procedure in the automotive context posses several challenges:

Long Lifetime

> "*Vehicles today are expected to have a development time of around three years, a production period of about seven years and a subsequent operation and service phase of up to 15 years. This results in a total product life cycle of around 25 years*" [Schäuffele and Zurawka, 2003]

Even with an average lifespan of ten years, road vehicles would still exceed almost all other electronic consumer product update-support-periods. Providing security updates over such a long time is a logistical and cost-intensive requirement. A software team has to collect vulnerability reports

and constantly patch the code base of many different components. This requires not only a team of experts but also servers for the safe distribution of these updates over a long time. Thus, OEMs have to ensure that each of their Tier-1 firmware suppliers is capable of updating the firmware during this time. This legacy support is especially cost intensive if the update process itself is not optimized for automotive stack-holder configurations. Additionally, software development methods may vastly change over time and a supplier may even go out of business during this phase.

Update Compatibility
Each firmware image has to be tailored specifically for one vehicle type in terms of functionality and diagnostic and other specifications. There need to be means of verifying that the new firmware is compatible with the current ECU firmware versions running on the target vehicle. Installing a non-compatible firmware version, in the worst case, could render the target vehicle inoperable.

Diverse Hardware
As described in 2.2.1, an automotive system consist of ECUs with a wide array of processing power and storage capacity. Implementing an update system, capable of handling the different resource and connectivity constraints of all ECUs, while retaining a certain security level is probably the most difficult challenge in this domain. The chosen ciphers enabling data integrity as well as authenticity between the stakeholders of the update process, therefore, become a trade-off between hardware costs and security.

Exchange Components
Due to the long lifetime and possible accidents, some electronic components may need to be exchanged over time. If ECUs connected by the in-vehicle bus-system are cryptographically tangled to each other to secure the update process, new hardware needs to undergo a series of key generation and key exchange procedures of some sort to restore authenticity before being able to function properly.

Update Procedure / Compromise Resilience
The update procedure has to accommodate the stack-holder configuration in the automotive industry. There are more stakeholders than in similar

update delivery configurations, their roles are much tighter interlinked and the potential damage is much higher. Given this, an automotive update procedure needs to ensure that the compromise of a single party does not endanger the whole update system (compromise resilience).

No dedicated standard specification
Automotive manufacturers are used to relying on legal requirements and standard specifications in many aspects of road-vehicle development. This provides manufacturers with kind of a frame and also guidelines on how to implement specific functions. There is currently no standard specification for cyber-security in automotive systems from one of the standardization organizations which addresses firmware updates in automotive systems and its before-mentioned challenges. In the last few years, ISO 26262 and SAE J3061 (described in section 2.5.2 and 2.5.3) were used to deal with cyber-security issues. As a consequence, cyber-security was dealt with as a functional safety requirement. The standard ISO/SAE 21434 (Road Vehicles — Cybersecurity engineering) 2.5.4 is currently under development, will be fully available in November 2020 [Schmittner, 2019], and focuses on cyber-security as a system requirement.

3.1 Problems With Existing Solutions

In this section, problems with the existing update solutions presented in 2 are discussed.

3.1.1 UPTANE

"*TUF and Uptane do not support verification of proper update installation on target devices.*" [Asokan et al., 2018]. Verification of proper update installation prevents drop-request and other attacks from being discovered. Thus, large-scale attacks on whole vehicle fleets are much more likely to remain undiscovered until the damage is inflicted.

Secondary ECUs which are using partial verification only verify and store

director metadata. If the director is compromised, mix-and-match attacks can be launched against these ECUs.

> "*A Mix-and-match attack is worse than a partial bundle installation or mixed-bundles attack because attackers can arbitrarily combine updates.*" [Kuppusamy et al., 2016]

This hardware-requirement to security trade-off for partial verification is disproportionate and bypasses the basic security concept TUF should provide.

3.1.2 ASSURED

ASSURED improves on TUF by adding attestation of proper update installation. The main problem with ASSURED as an automotive update system is that it is not specifically designed for that, but rather as a firmware update system for IoT devices. Consequently, no dependency check on the software configuration of different devices is performed. However, mapping these software dependencies must be an inherent part of an automotive update system (see *Update Compatibility* in chapter 3). Additionally, it requires an authenticated channel between the Domain Controller and an ECU, which is not a realistic requirement for automotive bus-systems.

3.2 Research Questions

RQ1: Which requirements must be fulfilled by an automotive system (and its components) to implement a secure and reliable update system? This question should clarify if specific hardware or software requirements are necessary in order to accommodate automotive update challenges or to meet the requirements of automotive standards.

RQ2: How can the system be integrated into existing automotive software? This question is important since diagnostic capabilities must not be affected by such a system. Hence, integrating an update system into a vehicle poses a significant challenge to automotive manufacturers.

RQ3: How can an update system function on a low-speed automotive bus-system? This question should clarify if even the lowest-speed automotive bus-system can be utilized in an automotive update system for state-of-the-art ECU hardware.

RQ4: How much memory is needed to store additional update specific content (metadata, keys, parameters,..)? This question is important because ECUs are often limited in their storage capabilities. Having a memory-efficient update system can often lead to micro-controllers with less flash-memory being used which would not have sufficient storage capabilities for the same functionality in other automotive update systems.

4 SOLUTION

In this chapter, the proposed update system and its procedures are described. First, the reference network-topology is discussed. Afterward, a few important identifiers/parameters are introduced, followed by the roles in the proposed update system and their purpose. At last, the update process itself and a few support processes are described. The system is defined in a way that it provides a certain degree of freedom in terms of implementation. A proof-of-concept with a reference implementation is given in chapter 5. An evaluation of how well the proposed update system addresses the problems stated in chapter 3 is performed in chapter 6.

4.1 Reference Network Topology

Every vehicle manufacturer has its own general in-vehicle network structures for its vehicle models. These structures differ depending on the field of application, general vehicle functions, and vehicle type.

What all modern in-vehicle networks have in common is a hierarchical network architecture, which is divided into multiple domains, each with its bus-system, managed by a domain controller (see Fig. 4.1). These domain controllers are interconnected through a high bandwidth bus-system e.g. via an automotive-ethernet switch along with the WVI (World-Vehicle-Interface) and optionally a diagnostic controller.

This common structure will be used as the reference network topology for this solution. This is the same architecture as shown in Fig. 2.1. Note that the solution can be adapted to fit an arbitrary network architecture as long as the chosen domain masters (see section 4.3.5) can request data from remote servers.

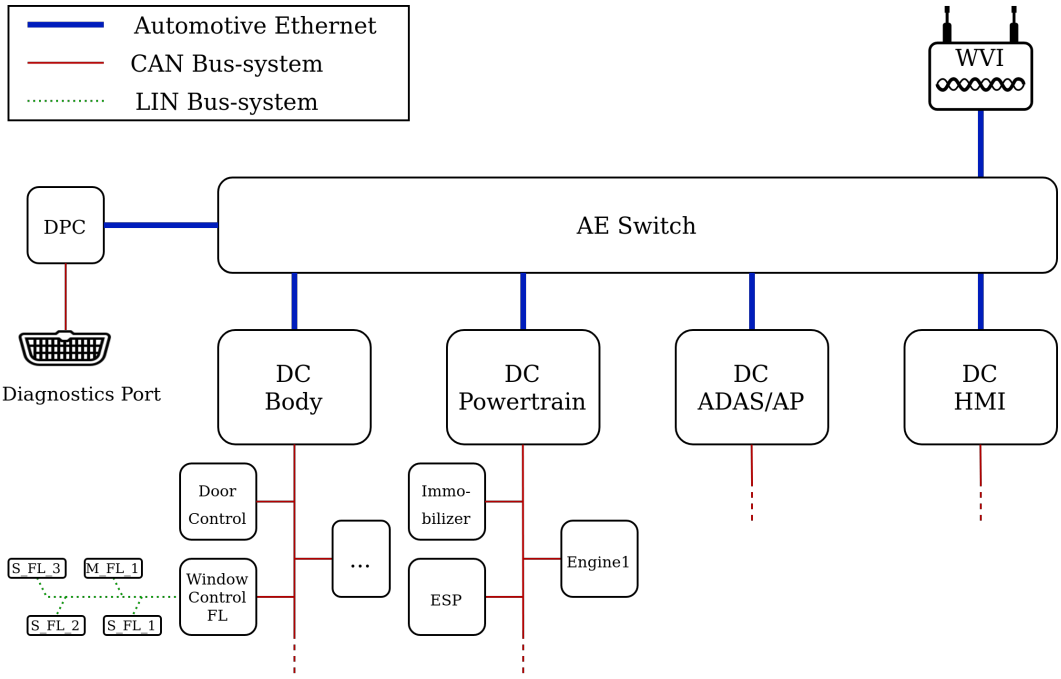

Figure 4.1: In-vehicle network topology in modern vehicles

4.2 Identifiers

In this section, three identifiers are defined which form the cornerstone of the proposed update system. Here is a short overview:

- `ECU_ID` uniquely identifies every ECU in entire the update system
- `TID` uniquely identifies the hard- and software configuration of one ECU
- `PID` uniquely identifies the hard- and software configuration of a dedicated inter-dependent hard-/software constellation.

Note that the coding sequence of an ECU, as well as the coding string of a vehicle, can be used as TID and PID respectively if their implementation fulfills the requirements for both identifiers stated below.

4.2.1 TID - Target Identifier

The Target Identifier (TID) uniquely identifies the hard- and software configuration of an ECU. ECUs which are identical in their hardware may perform different software functions. ECUs with identical hardware and identical software functions (same coding sequence) have the same TID. ECUs with the same TID and same software version (TIDversion) are identical and interchangeable in any vehicle regardless of type.

4.2.2 PID - Package Identifier

Vehicles of the same type often use the exact same ECU-Hardware but differ in the enabled overarching software functionality, while vehicles with slight hardware variations often use modular in-vehicle network domains which may be arbitrarily combined to form the basic EE-architecture of separate vehicle models. To map these ”software-functionality-domains”, Package Identifiers are used.

The Package Identifier (PID) uniquely identifies the hard- and software configuration of a dedicated inter-dependent hard-/software constellation, a network segment, or a software-functionality-domain in a road-vehicle. A vehicle's bus-system architecture is split up into independent logical network segments with a dedicated PID per segment. These logical ECU-groups are called `PID-domains`, see section 4.2.3.

The overarching software functionality of ECUs in different PID-domains must NOT be dependent on the version of the software executed by these ECUs, otherwise, compatibility issues, which may even render the vehicle inoperable, could be caused. PID-domains are therefore selected in a way that they interact as little as possible, may possess separate Diagnostic Specifications (DIDs, DTCs, etc.), and are present in multiple vehicles.

One PID for a whole vehicle is only necessary if the vehicle can not be split up into multiple PID-domains. Note that in this case, only vehicles

with identical hardware and the same executed software functions and versions (same coding string) may have the same PID.

4.2.3 PID-domains

PID-domains are defined by the contained ECUs, which all store the same PID. Ideally, PID-domains are chosen in a way that each network segment, e.g. each CAN-network in Fig. 4.1, represents one PID-domain. PID-domains can be nested or cascaded. For example, each CAN-network in Fig. 4.1 may represent one PID-domain, while all ECUs in the AE-network also form one PID-domain. Since ECUs in one PID-domain are not dependent on the software versions of ECUs in other PID-domains, `BECAUSE` can be combined with other update systems as well.

Splitting the in-vehicle network up into independent logical segments not only allows for easier update-ability but also reflects the modularity of in-vehicle networks in the automotive industry, in terms of both hardware and software. Few cars are developed completely from scratch. Many sub-networks from previous vehicles may be reused by the system architects as long as these fulfill all requirements for the new vehicle. This is mainly a monetary decision by automotive manufacturers.

4.2.4 ECU_ID

Every ECU in the update system has a unique ECU_ID. ECU_IDs are needed in order to identify a single ECU in a vehicle since the TID may not be sufficient. There may be two ECUs with the same TID in the same PID-domain but different TIDversion, e.g. for redundant systems. The ECU_ID does not have to be a dedicated parameter but may be derived from the unique serial number of an ECU. The problem with such derived values is that every Tier-x supplier may use different derivation methods or schemes, which eventually yield two ECUs from different manufacturers with the same ECU_ID.
A solution would be for an OEM to generate ECU_IDs for all ECUs in their

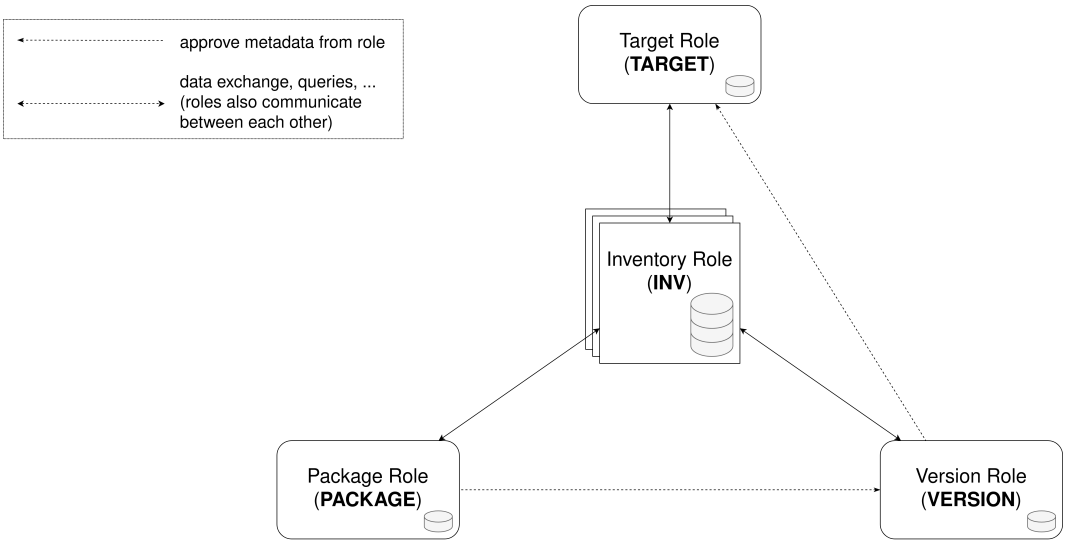

Figure 4.2: Roles and chain-of-trust outside the vehicle

vehicles and flashing them onto each ECU together with the bootloader. Ideally, the ECU_ID is stored in Read-only-Memory since it will and must never change.

4.3 Roles

In the following sections, the different roles are described. These roles consists of a set of tasks (e.g. signing metadata, verifying signatures,..) which should be performed by different devices in dedicated locations. For example, the domain-master role must be implemented in the vehicle, while roles which sign metadata should run on publicly accessible servers.

The Target-Role, the Package-Role, and the Version-Role are used to sign different types of metadata in order to split up the responsibility (divide-and-conquer principle) and gain compromise-resilience. These roles should therefore be performed on physically separated servers. The roles outside the vehicle are depicted in Fig. 4.2.

4.3.1 TARGET - Target Role

The Target Role signs metadata of actual software-images. Target-Metadata is used to verify the integrity and authenticity of a software image. For each software image, one target-metadata file is created which also contains additional information like type, version and the byte-size of the software image.

4.3.1.1 TARGET-Metadata

+ TID
+ TIDversion
+ bytesize(image)
+ compression algorithm of image
+ hashing algorithm of image
+ hash(image)

In this section, the parameters contained in a Target-Metadata file are described:

- **TID** uniquely identifies the hard- and software configuration of one ECU. For further information see section 4.2.1.
- **TIDversions** is the software version of a software image with corresponding TID. When new software is added to the update system, the corresponding TID is assigned and the current software version for this TID is increased and added to the newly created metadata.
- **bytesize(image)** denotes the size of the (compressed) software image in bytes.
- **compression algorithm of image** denotes the compression algorithm used to shrink the software image in size.
- **hashing algorithm of image** denotes the hashing algorithm used to compute the hash of the software image.
- **hash(image)** denotes the hash of the software image. This hash is used to verify the integrity of the corresponding software image.

4.3.2 PACKAGE - Package Role

The Package-Role signs metadata which describes the architectural structure of a PID-domain. This metadata is used by the domain master to verify for which ECUs it is assigned as domain-master.
The Package-Role also co-signs the Version-Metadata-Hash for the Version-Metadata-Verification, see 4.6 for more information.

4.3.2.1 PACKAGE Metadata

+ PID
+ PIDversion
+ updatePriority
+ hash(VERSION-Metadata)
+ hashing algorithm of VM
+ ECU_ID (multiple)
    - TID
    - ECU_ID of DM

In this section, the parameters contained in a Package-Metadata file are described:

- **PID** uniquely identifies the hard- and software configuration of a dedicated inter-dependent hard-/software constellation, a network segment, or a software-functionality-domain. For further information see section 4.2.2.
- **PIDversion** is the version of the corresponding PID. PID and PIDversion together uniquely identify each PACKAGE-Metadata. Note that the new TIDversion is increased by ”one step” in order for the ECU to install only consecutive versions (see section 4.6).
- **updatePriority** describes the type of update in terms of urgency. An example would be: service (low urgency), functional (medium urgency) or security (high urgency).
- **hash(VERSION-Metadata)** is the hash of the matching VERSION-Metadata (VERSION-Metadata for the corresponding PID and PIDversion).

- **hashing algorithm of VM** denotes the hashing algorithm used for `hash(VERSION-Metadata)`
- **ECU_ID entries** are a complete list of all ECUs in the vehicle which belong to the same PID-domain. Each entry states the `ECU_ID` and `TID` as well as the `ECU_ID of the corresponding DM` (domain master).

4.3.3 VERSION - Version Role

The Version-Role signs metadata which describes the changes in TIDversions of all ECUs with the stated PID between two consecutive PIDversions. ECUs which have no TIDversion change between the stated PIDversion in the VERSION-Metadata and the previous PIDversion, are not listed in the VERSION-Metadata. Therefore, the VERSION-Metadata is, on average, not only smaller than the PACKAGE-Metadata for the same PID, but also contains the same ECU entries per PIDversion for every vehicle with the same PID. Hence, the hash of VERSION-Metadata for a set of PID and PIDversion is always the same.
The only exception is the vehicle assembly process and the ECU replacement process, where special Full-Version-Metadata for a specific PIDversion is generated. Full-Version-Metadata includes an ECU-entry for every ECU in the PID-domain.

4.3.3.1 VERSION Metadata

+ PID
+ PIDversion
+ hashing algorithm of TM
+ number of ECU entries
+ ECU_ID (multiple)
    - TID
    - TIDversion
    - hash(TARGET-Metadata)

In this section, the parameters contained in a Version-Metadata file are described:

- **PID** uniquely identifies the hard- and software configuration of a dedicated inter-dependent hard-/software constellation, a network segment, or a software-functionality-domain. For further information see section 4.2.2.
- **PIDversion** is the version of the corresponding PID. PID and PIDversion together uniquely identify each PACKAGE-Metadata. Note that the new TIDversion equals the old TIDversion +1 in order for the ECU to install only consecutive versions (see section 4.6).
- **hashing algorithm of TM** denotes the hashing algorithm used for every hash of the corresponding TARGET-Metadata for all ECU entries in the VERSION-Metadata.
- **number of ECU entries** denotes the number of ECU entries in this metadata.
- **ECU entries** are a list of all ECUs in the vehicle which belong to the same PID-domain and which have changes in TIDversion between the previous PIDversion and the PIDversion stated in this VERSION-Metadata. Each entry holds the `ECU_ID`, `TID`, `TIDversion`, and the hash of the TARGET-Metadata for the stated TID and TIDversion combination.

4.3.4 INV - Inventory Role

The Inventory Role (INV) holds information about:

- every ECU and their current software configuration,
- the mapping of these ECUs to the vehicles in which they reside
- the mapping of ECU configurations to vehicle configurations (PIDversion-TIDversion mappings).
- the software images which were added to the system (and therefore already signed by TARGET)
- the ECU_SKEYs of every deployed ECU

Hence, it functions as a central data-hub for the update system. There are already solutions on how to implement such a database system in a compromise resilient way as [Kuppusamy et al., 2016] pointed out. Since INV does not sign any metadata, it can for example be implemented as an instance of The-Update-Framework (TUF) on its own. It may be advantageous in some situations to split up the Inventory Role into sub-roles in order to delegate vehicle feature configuration and management to different stakeholders. An exemplary implementation is given in chapter Proof-of-Concept, see 5.

Regardless of the concrete implementation, compromising this role should at least be as difficult as compromising TARGET, PACKAGE, and VERSION conjointly.

4.3.5 DM - Domain-Master Role

The Domain Master Role represents a middleman between the ECU to be updated and the update servers. It is not a dedicated device, but rather a function, similar to the ”primary ECU”-Role in the ASSURED update system [Asokan et al., 2018].
DM fetches metadata and images from update servers, validates both and distributes it through the local bus-system to the Target-ECUs.
Each Domain Master is responsible for all ECUs of one PID in the vehicle. In modern hierarchical in-vehicle network topologies, domain controllers are ideal for this task since they contain enough memory and computational performance, act as a direct gateway for all ECUs with their associated PID,

and are bus-masters in master-slave bus-systems.

Depending on the in-vehicle network architecture, there may be some cases in which the task of the Domain Master Role may be executed on a device other than an actual domain controller. This device needs enough memory in order to buffer software images and metadata and the computational capability to verify all metadata in a reasonable time. In a master-slave bus-system, the Domain Master Role should always be executed by the bus-master.

4.4 ECU Memory Content

Independent of the hardware and the concrete software implementation, each ECU holds the content depicted in Fig. 4.3. In order to keep the software execution during vehicle runtime independent from the update process, a (second-stage-)bootloader is used.
Instead of running the software image responsible for the ECU function directly, the code-execution starts at this custom "update-bootloader"[1]. Its task is to handle everything related to the ECU update process e.g. fetching new images, verifying metadata, comparing the hash of an image with its hash from the TARGET-Metadata, and handling over execution to the images. Two image spaces are used for this solution. Hence, if an error occurs during the update process (e.g. the storage operation for the new image), the currently used image is not affected.

An exemplary implementation is given in chapter 5.

Each ECU memory contains the following static data:

- **ECU_ID** is described in 4.2.4.
- **ECU_SKEY** is the unique symmetric key of the ECU. It is the only parameter stored by the ECU which is secret.
- **TID** is described in 4.3.1.

[1]Note that, depending on the embedded system and its SoC, a primary bootloader, e.g. located in ROM, may be executed prior to the second-stage-bootloader execution from flash memory, in order to initialize hardware components.

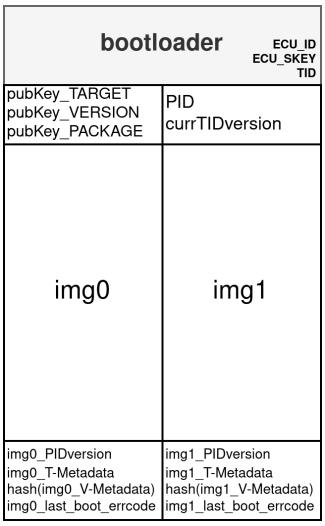

Figure 4.3: Simplified ECU Memory Content and Layout

Each ECU memory contains the following dynamic data:

- **pubKey_TARGET**, **pubKey_VERSION**, and **pubKey_PACKAGE** are the public keys of the corresponding roles which are used to validate the TARGET-, VERSION- and PACKAGE-Metadata.
- **PID** is described in 4.2.2.
- **currTIDversion** is the current TIDversion. This value is used to determine which image should be started.

Each ECU memory contains the following dynamic data per image:

- **PIDVersion** is the PID-Version associated with the image.
- **last_boot_errcode** states whether the last boot of the corresponding image was successful and may hold the resulted error-code if it was not.
- **T-Metadata** is the complete TARGET-Metadata of the associated image.
- **hash(V-Metadata)** is the hash of the VERSION-Metadata of the associated image. It is used to verify a PID/TIDversion switch. Note that the hashing algorithm needs to be predefined so that every ECU in the PID domain can verify the same signed hash-value (see section 4.6).

4.5 Supplementary Procedures

In this section, exemplary realizations of supplementary procedures for the update system are defined. Note that these procedures are intended to act as exemplary solutions for the most relevant processes which need to be performed apart from the update process itself.

4.5.1 Initial ECU Flashing

Prior to the vehicle assembly, each ECU is flashed with a bootloader, the current public role keys, its ECU_ID, TID and ECU_SKEY. At the time of flashing, it may not be clear in which vehicle and in which PID domain the ECU may later reside in. The ECU_SKEY is generated by INV (in order to prevent key collisions) and delivered to the flashing process in a secure and authenticated way.

Every ECU which is not a DM, is primed with PID, PIDversion and TIDversion of zero. Every ECU which is a DM is primed with a TIDversion and PIDversion value of zero.

4.5.2 Vehicle Assembly

During the assembly process of the vehicle, each mounted ECU is registered and the ECU-VIN relation is stored in the Inventory database system along with zero-values in its TIDversion and PIDversion fields. If the ECU is not a domain master, the PID value in the database is also set to zero. At the end of the assembly line, each vehicle is booted up. The bootloader on every ECU detects the zero values in the version fields and the PID-domains switch into update mode.
Furthermore, the zero values indicate to the Inventory Role, the Version Role (via the ECU-Manifest) and the individual bootloaders, that the newest PIDversion instead of the consecutive PIDversion has to be installed.
For these initial updates, VERSION generates Full-VERSION-Metadata,

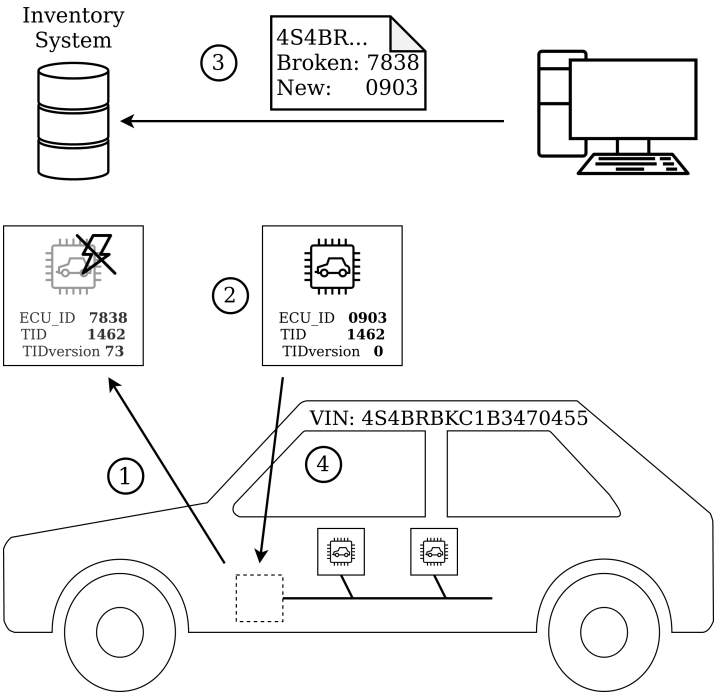

Figure 4.4: ECU replacement process

containing all entries for all ECUs of one PID[2]. This prevents multiple consecutive updates for each vehicle compared to vehicles with ECUs which would already hold images with a specific TID- and PIDversion, since these versions may already be many versions behind the current TID- and PIDversion when being mounted during vehicle-assembly.
This ”initial-update-method” is also very convenient and efficient for the vehicle manufacturer since the update process can start as soon as every ECU is mounted and connected to the vehicle. No diagnostic equipment has to be wired to every single vehicle as in usual coding and assembly processes.

[2]If the VERSION-Metadata for this initial update would be too large for an ECU to store in its memory, e.g in its SRAM, it may utilize the second image area in its flash-memory to buffer the VERSION-Metadata.

4.5.3 Replacing ECUs

During the lifetime of a vehicle, some ECUs may break due to various reasons. The following steps, see Fig. 4.4, shall act as an example of how this process can be performed:

① The faulty ECU is removed from the vehicle in an authorized workshop
② An ECU with the same TID is chosen as a replacement
③ The mechanic authenticates itself to INV and creates a hardware change request (e.g. via a web-interface) for the vehicle including the VIN, the ECU_ID of the broken ECU, and the ECU_ID of the replacement ECU.
④ The mechanic replaces the ECU.

At the next vehicle start, the ”zero”-values of TIDversion and PIDversion in the memory of the replacement ECU indicate to its bootloader that it needs to perform an update. The corresponding metadata for the current PID and PIDversion is sent to the ECU and the ECU installs it. Note that in this case the Full-VERSION-Metadata for the current PIDversion is used. As soon as the ECU confirms its successful update to the current PIDversion, the hardware change request is closed and the new ECU is added to the INV database system.

4.5.4 Renewing Public Role Keys

Each ECU regularly (e.g. before every update) requests a public key statement from INV, signed with ECU_SKEY. In this statement, the roles and matching current Key-IDs are listed. If they differ from the Key-IDs currently stored by the ECU, the ECU fetches the related new public keys directly from the corresponding roles.
In order to prevent replay attacks (show an ECU outdated keys), the public key statement also contains the value of a counter. This counter-value is managed by INV, unique for every ECU, and increased by one after every public key statement request.

4.6 Update Process

Here the individual steps of the complete update process are described. It is the most important part of this solution and is therefore described in detail. In order to maintain readability, the process description was split into two parts. Note that this solution was developed with the cryptographic algorithms in mind, which were also used for the proof-of-concept implementation (see section 5.2.4). However, this does not mean that the solution is limited to these algorithms and cryptographic principles by any means. This is also the reason no dedicated algorithms are proposed to be used in this section.

4.6.1 Part A - Fetching Data

(A1) DM requests the ECU-Manifest from every ECU in its Domain. This ECU-Manifest includes:

- `ECU_ID`
- `TID`
- `TIDversion of primary image`
- `TIDversion of secondary image`
- `IV` (Initialisation Vector)
- `HashingAlgorithm for the signature`
- `EncryptionAlgorithm for the signature`
- sign(`ECU_SKEY`; hash(`ECU_ID`, .. ,`TIDversion of secondary image`))

When booting, the bootloader of an ECU boots the image those TIDversion in the TARGET-Metadata equals the currTIDversion. This image is called the primary image. The other image is called the secondary image. If the metadata for an image is invalid, the value of `TIDversion` for this image is set to zero.
`IV` (Initialisation Vector) denotes random bytes acting either as the initialisation vector for a symmetric encryption algorithm like AES or are just added for more randomness in the input data for the signature scheme. sign(`ECU_SKEY`; hash(`ECU_ID` - `TIDversion of secondary image`)) denotes the hash of the data from `ECU_ID` up to

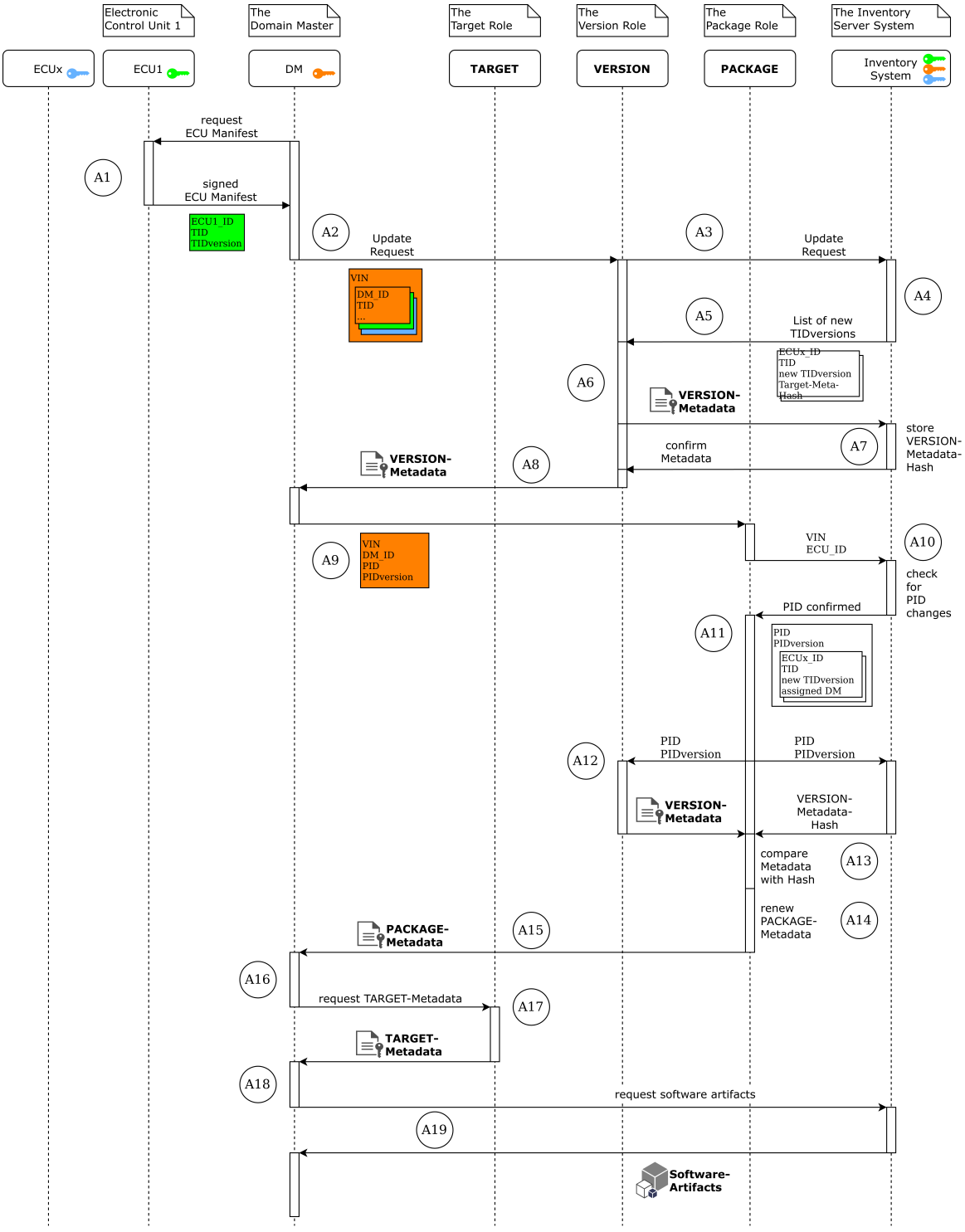

Figure 4.5: Depiction of the proposed Update Process (Part A)

TIDversion of secondary image (or up to IV, depending on the implementation) in the ECU-Manifest, signed by the ECU_SKEY.

(A2) After receiving every ECU-Manifest, DM adds the VIN and its own ECU-Manifest to this ECU-Manifest-List signs it with its own ECU_SKEY, and sends this Update-Request to VERSION

(A3) VERSION contacts INV and relays the received data to it.

(A4) INV checks the following for every ECU-Manifest:

- every ECU-Manifest is signed correctly by the corresponding ECU
- ECU_ID and TIDversion are the same as in its database for every ECU
- the DM signed the whole package correctly
- every ECU in the ECU-Manifest-List belongs to the same PID
- every ECU_ID is part of the same VIN
- no ECU is blacklisted (e.g. ECUs from a vehicle which was reported as stolen)
- the ECU_ID of DM is marked as DM in its database. If received data and data from the INV database differ, INV checks for hardware change requests for this VIN and may update its database. For further information on ECU replacement, see section 4.5.3.

(A5) INV evaluates the consecutive PIDversion of the current PIDversion.
INV creates a list of the newest compatible software versions for each TID of this PID from its database.
INV drops TID entries where the currently installed software version equals the software version to be installed and adds the corresponding target-meta-hash for every remaining entry.
INV sends this list to VERSION.

Note:
If every update request from a vehicle containing the same PID-domain would lead to the creation of a new PIDversion, INV would soon run out of PIDversion numbers. To prevent this, INV regularly creates new

PIDversions for each PID and adds entries to its table. The software changes between two consecutive versions should not be to small in order to prevent constant updates and not to big in order to perform updates in a reasonable time (e.g. a few megabytes per PIDversion, depending on involved bus-systems).

(A6) VERSION uses this list together with PID and PIDversion to create new VERSION-Metadata, see 4.3.3. Therefore, VERSION-Metadata contains only ECU entries where a newer software version exists. If the PID is very common, VERSION may have already created and stored the signed VERSION-Metadata.
VERSION signs this VERSION-Metadata.
VERSION sends signed VERSION-Metadata to INV.

(A7) INV confirms that the signed VERSION-Metadata is valid and adds a hash of the signed VERSION-Metadata to its database. INV creates an entry in its pending update table and adds an entry for every ECU which should be updated to it.
An update-approach is chosen where the current version and the new PIDversion are consecutive. Hence, ECUs in a vehicle with the same PID have to install one PIDversion after the other. The VERSION-Metadata contains the same ECU entries per PIDversion for every vehicle with the same PID. Hence, the VERSION-Metadata hash for one PIDversion is always the same. Otherwise, there would be one VERSION-Metadata(-hash) for every possible leap between two versions of the same PID The only exception to this rule is the full-version-metadata defined in section 4.5.2.
INV regularly creates new PIDversions (defines which TIDversions belong to which PIDversion) and adds entries to its table. The software changes between two consecutive versions should not be too small in order to prevent constant updates and not too big in order to perform updates in a reasonable time.

(A8) Upon receiving a confirmation from INV, VERSION sends the VERSION-Metadata to DM.

(A9) Upon receiving the VERSION-Metadata, DM sends:

- VIN
- ECU_ID of DM
- PID
- PIDversion

to the Package-Role (PACKAGE).

(A10) PACKAGE contacts INV with the received VIN and `ECU_ID` of the DM. INV checks if the given `ECU_ID` really belongs to the DM of the given VIN and confirms the PID to PACKAGE.
If the given PID and PID in the database differ, INV may check for pending PID change requests. This could happen if the vehicle owner purchased premium software functionality which changed the PID of (at least one domain in) the vehicle.
Note that PID change of a PID-domain is not defined in this solution, but discussed in section `Future Work` in the evaluation, see 6.3
If given `ECU_ID` and VIN do not match, INV may check for pending ECU change requests. This may happen if the original ECU with DM functionality broke. In this case, a workshop would have to replace the broken ECU with a new one (with or without the same TID) and create a hardware change request for the INV system.

(A11) PACKAGE requests from INV:

+ PIDversion
+ a list of
    - ECUx_ID
    - TID
    - new TIDversion
    - assigned DM

for the given PID

(A12) PACKAGE requests Version-Metadata for given PID and PIDversion from VERSION

(A13) PACKAGE cross-checks Version-Metadata with the help of INV (com-

pares the hash from INV with the hash of VERSION-Metadata)

(A14) PACKAGE renews the current PACKAGE-Metadata for the corresponding PID by renewing

- the hash of the approved VERSION-Metadata
- the PIDversion of the current VERSION-Metadata

and signs it.

(A15) PACKAGE sends signed PACKAGE-Metadata back to DM.

(A16) DM compares TID entries of VERSION- and PACKAGE-Metadata and checks if all TIDs from VERSION-Metadata exist in PACKAGE-Metadata as well. DM checks if the hash of the current VERSION-Metadata in PACKAGE-Metadata is the same as the hash of the actual VERSION-Metadata.

(A17) DM requests the TARGET-Metadata for all TIDs from the ECU entries in package metadata where the TID also exists in the VERSION-Metadata and where it is listed as responsible DM in the PACKAGE-Metadata.
Additionally, DM may hold a list of TIDs in its domain for comparison.

(A18) DM checks if the received TARGET-Metadata is valid and requests the related software images.

(A19) TARGET sends the requested images to DM. Depending on the number of different Coding Sequences, some images may have been deleted since the related metadata creation. Therefore some images may have to be newly created from a base image.

4.6.2 Part B - Distributing Data

All steps in Part A can be performed independent of the vehicle state. Depending on `updatePriority` stated in the PACKAGE-Metadata, different

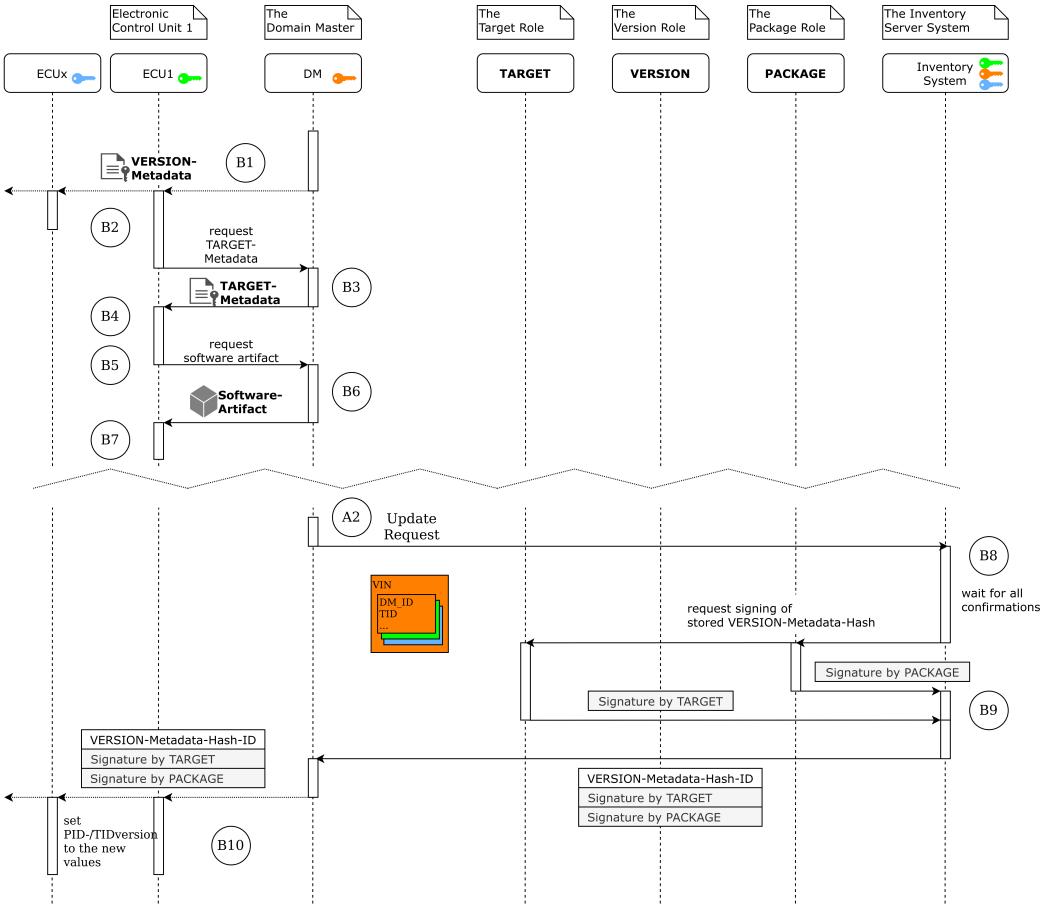

Figure 4.6: Depiction of the proposed Update Process (Part B)

update approaches may be taken in Part B. The vehicle may switch automatically to a dedicated update-state after ignition if security updates need to be applied (distributed through vehicle bus-systems). It can also be possible to give the user the option to actively enter update-mode in order to manually start the update process. If a pending update is not initialized manually, the bootloader on every ECU in each PID-domain will fetch new updates from its DM at the next reboot.

Due to many individual software-images, in-vehicle update distribution may take more than a few seconds. In this case, a dedicated update mode

should be used prior to vehicle start-up. Since this delays the vehicle start-up, a confirmation for the update distribution should be retrieved from the vehicle operator. It is essential that every PID-version contains only small changes in order to keep the total amount of updates per start-up as low as possible.

(B1) DM broadcasts the received VERSION-Metadata to its whole PID-domain.

(B2) Each ECU in the Domain receives the VERSION-Metadata.
Each ECU validates the VERSION-Metadata.
Each ECU checks if its own TID is in the VERSION-Metadata.
If so, the ECU requests/awaits TARGET-Metadata for its TID, otherwise, the ECU is ready to exit the bootloader.

(B3) DM sends out requested TARGET-Metadata to the ECUs with corresponding TID.

(B4) Each ECU validates TARGET-Metadata with its public key and checks the following:

- The stored TID equals the TID in the TARGET-Metadata
- The stored PID equals the PID in the VERSION-Metadata
- The TIDversion in the VERSION-Metadata is the same as in the TARGET-Metadata and if it is newer than the current version
- The size of the new software image does not exceed its own storage capabilities.
- The `hash(TARGET-Metadata)` in the VERSION-Metadata is the same as the actual hash of the TARGET-Metadata

(B5) The ECU stores a hash of the VERSION-Metadata, discards the VERSION-Metadata, and requests the corresponding software image from DM.

(B6) DM sends the software image to the ECU.

(B7) The ECU receives the software image until `bytesize(image)` and writes it to its flash memory. The ECU validates the software image by comparing `hash(image)` in TARGET-Metadata with the actual hash

of the image. The ECU stores the TARGET-Metadata along with the software image.

(B8) The next ECU-Manifest requested by DM and sent to INV contains the TIDversion of the new image. INV waits for all ECU-Manifests of the ECUs listed in its pending update table entry for this PID-domain in the vehicle.

(B9) If all ECU-Manifests confirm that the ECUs in the PID-domain have successfully installed the images of the new PIDversion, INV links the stored VERSION-Metadata-Hash to the ECU entries with a confirmed update and request TARGET and PACKAGE to co-sign this Hash. Both roles check if all ECUs confirmed proper update installation, through a database query on INV, and then both sign the Hash. The signatures are appended to the first **x**-bytes of the VERSION-Metadata-Hash (VERSION-Metadata-Hash-ID) to form the Version-Metadata-Verification (VMV). **x** can either be the total number of bytes of the VERSION-Metadata-Hash or a value lower than that, which is still long enough to prevent hash-collisions with other VERSION-Metadata-Hashes. E.g. for the proof-of-concept implementation, **x** was chosen to be 8-bytes.

(B10) This VMV is then broadcasted to the corresponding domain in the vehicle (e.g. DM may regularly query INV for the VMV). After receiving the confirmation, each ECU sets its `currTIDversion` to the new TIDversion and its `currPIDversion` to the new PIDversion. At the next boot, the signed ECU manifests with the new TID-/PID-versions listed, confirm to INV that the version change was successful. If all ECU manifests confirm the change, the pending update entry is closed and the version-numbers are set to the new versions in the database of INV.

When booting, the bootloader of an ECU boots the image those TIDversion in the TARGET-Metadata equals the currTIDversion. This image is called the primary image. If an image exits correctly (returns 0 to bootloader), the bootloader sets `last_boot_errcode` of this image to zero. The Bootloader is consequently able to detect problems, request updates, and report errors.

5 PROOF-OF-CONCEPT

In this chapter, the practical implementation of the presented update solution is described. The individual hardware and software of the components as well as their interplay is discussed.
The goal of this proof-of-concept implementation is to demonstrate the practical viability of the presented update system as well as provide means to run benchmarks on the update process itself. The focus was therefore on the in-vehicle network part of the update process.

5.1 Hardware

In the reference network topology (see 4) the domain controllers, which run the software functions of the domain master role, communicate with the World-Vehicle-Interface (WVI) over a high-bandwidth bus-system (e.g. automotive ethernet). The WVI provides access to the internet via its cellular network module.
Data exchange between the domain master role and the roles outside the vehicle can therefore utilize high-level network protocols like IP and is therefore trivial. Furthermore, fetching new software images and metadata can be done by the domain master role at any time regardless of vehicle state. Hence, the communication between a domain master and the roles outside the vehicle does not directly influence the speed by which updates can be distributed by the domain master to its associated ECUs.

Due to this, only the functions of the two main roles in the in-vehicle network architecture were implemented: An ECU and a Domain Master (see Fig. 5.1).

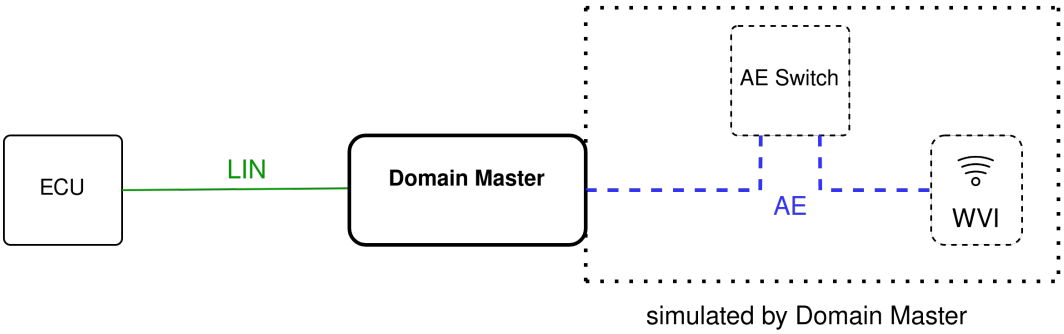

Figure 5.1: In-vehicle network topology for the proof-of-concept implementation

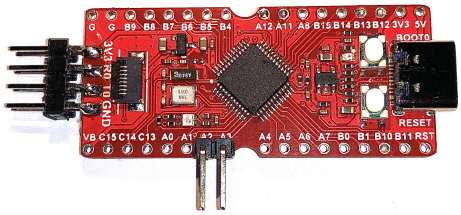

Figure 5.2: Sipeed Longan Nano

In the following sections, the individual parts of the proof-of-concept setup and the reasons for them being used are described.

5.1.1 The ECU (32-bit RISC-V)

As described in 2.2, the variety of hardware used for ECUs is as diverse as their functions. For this proof-of-concept, a **Sipeed Longan Nano development board** was used (see Fig. 5.2). It contains a GD32V microcontroller unit from GigaDevices with a 32-bit RISC-V kernel of type RV32IMAC and comes with an additional 160x80 pixel OLED-Display. The exact name of the MCU is GD32VF103CBT6, which features a maximum clock frequency of 108Mhz, contains 128kB of Flash Memory and 32KB of SRAM while consuming only a third of the power of comparable ARM Cortex-M3 microcontrollers.

This open standard instruction set, the integrated CAN-bus interface of

the GD32V the OLED-Display (which was used for testing purposes of the update system) were the main reasons this development board was chosen.

5.1.2 LIN bus

Although the GD32V features a CAN-bus interface, LIN was chosen as the bus-system for ECU-to-DM communication. As discussed in 2.3, LIN is by far the bus-system with the lowest-bandwidth used in modern vehicles.
If the proposed update system can deliver updates over LIN, any currently used automotive bus-system can be used since bandwidth as the limiting factor is no issue in this case.

5.1.3 Domain Master (32-bit ARM11)

As stated in 4.3.5, domain controllers in modern vehicles are equipped with sufficient computational capabilities to perform the tasks of a domain master role[1]. Hence, and since the device acting as domain master in this implementation should also partially simulate tasks of roles located outside the vehicle, a RaspberryPi 1B[2] single-board computer was chosen.
Featuring a Broadcom BCM2835 with a 32-bit RISC ARM11 core operating at 700Mhz, this device is quite similar to what can be found in common low-performance domain controllers.

5.2 Software

As development environment for the Sipeed Longan Nano development board, PlatformIO IDE in combination with the Atom Text-Editor was used

[1] e.g. Tesla uses an Intel Atom E8000 for its MS, MX MCU2, and M3 MCU, see https://tesla-info.com/blog/technical-hardware-differences.php

[2] for more information see https://www.raspberrypi.org/products/raspberry-pi-1-model-b-plus/

(see PlatformIO-IDE for Atom).
After installing the Atom-Text-Editor, the package `platformio-ide` can be installed directly through Atoms internal package repository. Several other Atom packages are thereby installed which also alter the graphical appearance of Atom. Following its installation, PlatformIO itself provides frameworks/toolchains which can be installed via its "PlatformIO Home"-tab in Atom. Support for the Sipeed Longan Nano is included in the `GigaDevice GD32V` package.

A new project was created with the following project configuration file (platformio.ini):

```
[env:sipeed-longan-nano]
platform = gd32v
board = sipeed-longan-nano
framework = gd32vf103-sdk

board_build.mcu = GD32VF103CBT6
board_build.f_cpu = 108000000L

upload_protocol = serial
```

Listing 5.1: Content of platformio.ini

The version numbers for the main software toolchain components are listed below:

- Atom Text Editor 1.50.0
- PlatformIO IDE 2.7.2
  - PlatformIO Core 5.0.1
  - PlatformIO Home 3.3.0
  - GigaDevice GD32V (PlatformIO Package) 1.1.2
    - RISC-V Gnu Compiler Toolchain 9.2.0

5.2.1 Domain-Master Implementation

The DM was running Raspbian GNU/Linux 10 with python3 (version 3.7.3) installed. The python packages `pyserial` and `crcengine` were added via the following bash commands:

```
sudo apt install python3-pip
sudo pip3 install pyserial
sudo pip3 install crcengine
```

Metadata-files, VMV-files, and software images would normally be fetched from the roles outside the vehicle and then stored in the file system of the DM. In this proof-of-concept implementation, the task of creating and adding these files to the filesystem was performed by individual scripts on the DM. These scripts were written with the Atom-Text-Editor mentioned above and did not require any additional development software. Hence, update-ability of the DM through the presented update framework was not implemented.

5.2.2 ECU Implementation

As depicted in Fig. 5.3, the ECU flash memory is divided into four parts:

- Bootloader
- Image 0
- Image 1
- Metadata

65kB are assigned to the bootloader. The current implementation is written in C (C11 standard) and takes up about 64kB. The reason for its size lies in the implementation of the ed25519 algorithm, which accounts for approximately 44kB (see 5.2.4) and the display driver (used for debugging purposes) which accounts for about 4.5kB.
30kB are assigned to each image.
1kB is assigned to the Metadata area, which contains all metadata for each image as well as the bootloader.

As depicted in Fig. 5.3, the bootloader metadata contains a parameter named `currTIDv`. This parameter holds the current TID version number (or zero after the flashing process) and therefore identifies the primary image[3].

[3]Primary Image: the primary image is the image which is currently in use, identified by currTIDv. The secondary image is an image which is currently not the primary image.

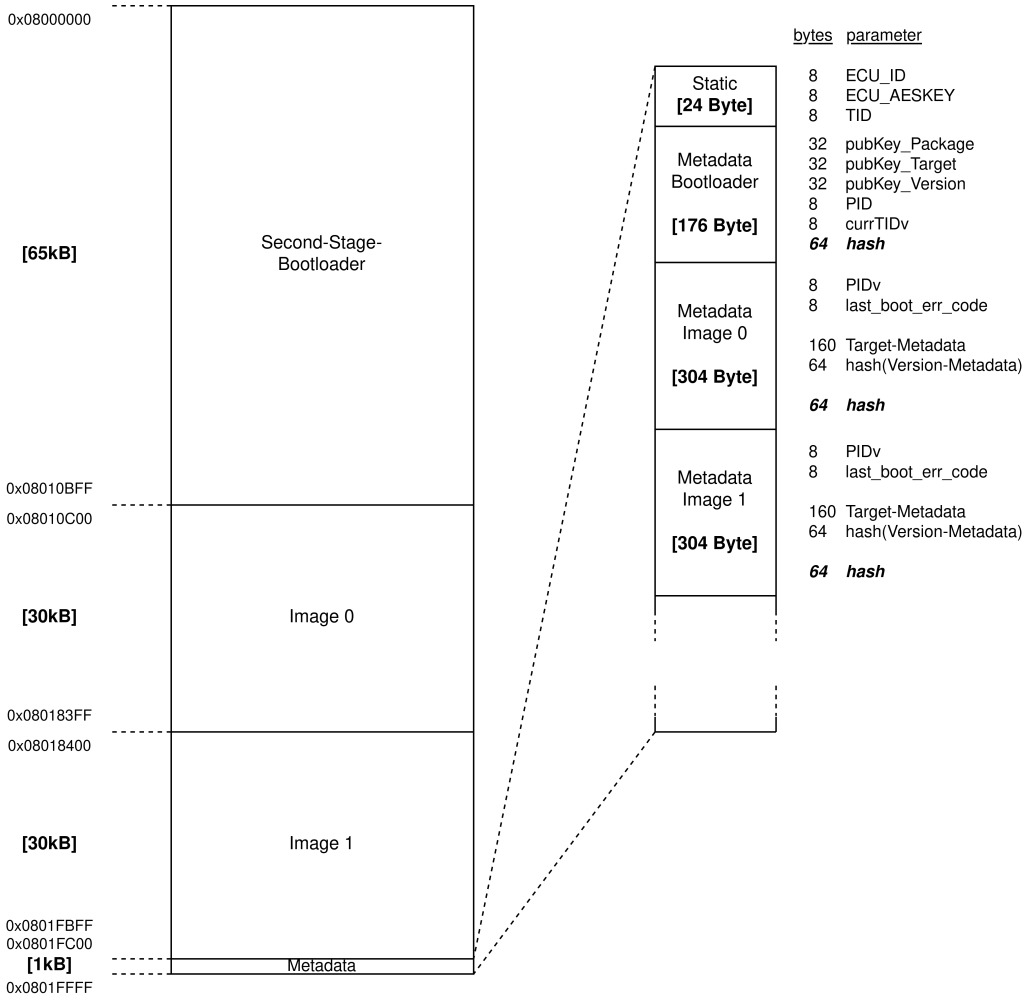

Figure 5.3: ECU Memory Content

Only an image with the same TIDv in its Target-Metadata can be the primary image and therefore being booted.

5.2.2.1 Metadata Storage

The Metadata memory section holds all metadata for the bootloader and the (two) images as well as ECU_ID, ECU_AESKEY, and TID (see Fig. 5.3).
C-Structs are used to handle these metadata in the implementation. One instance of 5.2 and two instances of 5.3 are filled with the content of the corresponding flash-memory area at the start of the bootloader execution. The 64 byte sha3-512 hash in both structs is used to verify the integrity of the memory area. This mechanism mainly aims at preventing errors due to memory corruption (e.g. bit-flips).

```
typedef struct __attribute__((__packed__)) {
  char pubKey_P[32];                    // 32   bytes
  char pubKey_T[32];                    // 32   bytes
  char pubKey_V[32];                    // 32   bytes
  uint64_t pid;                         // 8    bytes
  uint64_t currTIDv;                    // 8    bytes
  uint8_t hash[64];                     // 64   bytes (512 bit)
}meta_bl;                               // 176  bytes (22x 64 bits)
```

Listing 5.2: Bootloader-Metadata implementation as c-struct type-definition

If the verification of the bootloader metadata fails, its content is reset to its initial state which was present after the flashing process. The corresponding values in the next sent ECU Manifest indicate to INV that the bootloader metadata of the ECU was reset and that it is not able to select the correct image to boot (due to currTIDv set to zero). In this case, several options for error handling are feasible (see 5.2.2.4).

```
typedef struct __attribute__((__packed__)) {
  uint64_t pidv;                        // 8    bytes
  uint64_t last_boot_err_code;          // 8    bytes
  tmeta tm;                             // 160  bytes
  uint8_t vm_hash[64];                  // 64   bytes (512 bit)
  uint8_t hash[64];                     // 64   bytes (512 bit)
}meta_img;                              // 304  bytes (38x 64 bits)
```

Listing 5.3: Image-Metadata implementation as c-struct type-definition

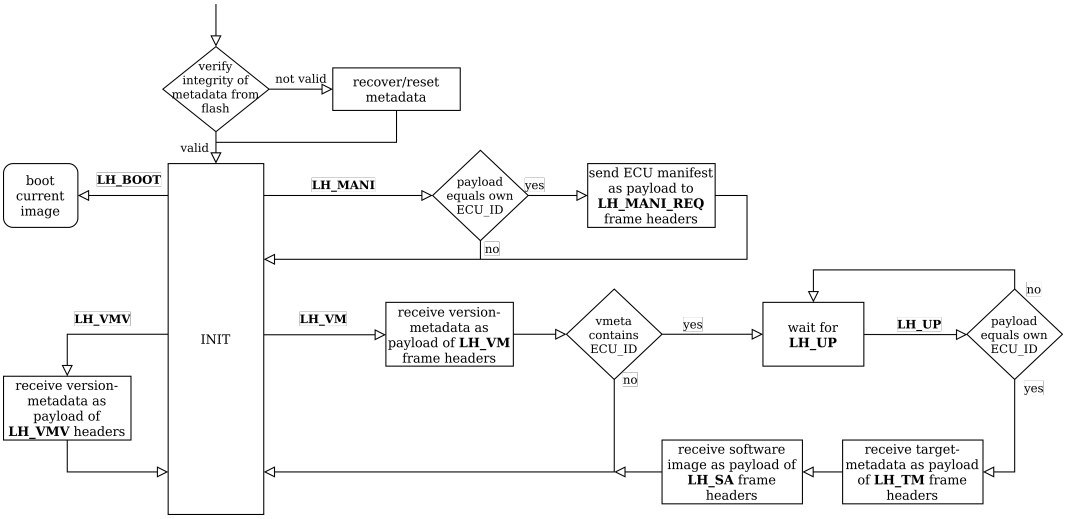

Figure 5.4: ECU bootloader state diagram

If the verification of one of the image metadata fails, the metadata is reset by setting its c-struct content to zero values (see Fig. 5.4). This renders the corresponding image non-executable due to its failing image verification and therefore unlocks it to be overwritten by the next update.

At the end of the bootloader execution, the (altered) c-structs are written back to flash-memory.

5.2.2.2 ECU State Machine

After the metadata has been loaded from flash-memory and verified, the bootloader goes into the INIT-state and busy-waits for one of four possible LIN-Frames. As described in 2.3.2, ECUs can only respond to LIN-Frame-Headers sent by the bus-master (the DM). The Frame-IDs are predefined and specify if the following LIN-Frame-Payload field contains payload or is meant to be filled with payload by the addressed ECU. All LIN-Frames during bootloader communication are set to be 8 bytes long.

All frames which were defined are listed below in the form [LIN-Frame-ID, LIN-Payload]:

- Manifest Request [LH_MANI, ECU_ID]
- Update Request [LH_UP, ECU_ID]
- Boot Broadcast [LH_BOOT, <not def.>]
- VM Broadcast [LH_VM, <not def.>]
- VMV Broadcast [LH_VMV, <not def.>]
- TM Packet [LH_TM, <bytes of target metadata>]
- SA Packet [LH_SA, <bytes of software image>]

The Frame Manifest Request indicates to the ECU with the given ECU_ID, that its Manifest is requested. The ECU then waits for further Frame Headers of type LH_MANI_REQ and fills the payload fields with its ECU-Manifest data. The frames are depicted in Fig. 5.8.
The Frame Boot Broadcast indicates to all ECUs that they shall boot their current image (if valid).
The Frame VM Broadcast indicates to all ECUs that the next frames with header LH_VM contain version-metadata (see Fig. 5.6). After completing the Version-Metadata broadcast, DM sends out Target-Metadata and software-images to the ECUs for which an ECU-entry exists in the Version-Metadata. These ECUs wait for an Update Request containing their ECU_ID as payload. Receiving such a frame indicate to the ECUs that the following frames with LIN-Frame-ID LH_TM and LH_SA are addressed to the ECU with the specified ECU_ID. Frames with Frame-ID LH_TM contain Target-Metadata as Frame-Payload (see Fig. 5.5). Frames with Frame-ID LH_SA contain the bytes of a software-image as Frame-Payload.

The Frame VMV Broadcast indicates to all ECUs that the next frames with header LH_VMV contain the version-metadata verification (see Fig. 5.7). If VMV is valid for the primary image, currPIDv of that image is increased by one and the stored version-metadata-hash of that image is set to zero. No further steps need to be taken since currTIDv in the bootloader-metadata already identifies the right image as primary.
If VMV is valid for the secondary image and TIDv of this image is equal to TIDv of the primary image plus one, the version-meta-hash of the secondary is set to zero, and currTIDv in the bootloader-metadata is set to the value of the secondary image, thereby making it the new primary.
If VMV is not valid, it is ignored and the bootloader switches back to INIT.

The implementation of Metadata, the Version-Metadata-Verification as well as the ECU-Manifest, their division into individual frames, and the individual data-types are described in 5.2.3.

5.2.2.3 External Software Sources

The only external source code for the ECU implementation, apart from the required modules from the `GD32VF103_Firmware_Library`[4], were the implementations of the cryptographic algorithms described in 5.2.4 and the driver for the display[5].

5.2.2.4 Error Handling

During the bootloader execution, several verifications may fail due to various errors. The Bootloader handles them by altering or not altering associated metadata parts. INV can detect, and even distinguish, between different errors by comparing the received ECU manifest with the expected manifest and the last received manifest.
INV may notify the OEM, who can then take further measures if necessary.

5.2.3 Metadata and Datatypes

In the following sections, the content of the generated (metadata-)files and other data-objects, the datatypes of the parameters within these objects as well as their transmission to the ECU are described.
All metadata-files contain their parameters appended to each other in binary format. Splitting these files up into 8-byte chunks consequently results in the payload of the LIN-frames of the corresponding metadata.

[4]see https://github.com/riscv-mcu/GD32VF103_Firmware_Library
[5]see https://github.com/sipeed/Longan_GD32VF_examples/tree/master/gd32v_lcd

5.2.3.1 Signature scheme

As depicted in Fig. 5.5, 5.6, 5.7, and 5.8, all metadata, the version-metadata-verification, and the ECU manifest contain at least one signature each. These signatures are all stored and transmitted in the same 72-byte long format.
The signatures in Fig. 5.5, 5.6, and 5.7 are public-key signatures, created with the respective private role keys. `sig_hash_algo` defines the hashing algorithm and `sig_enc_algo` the encryption algorithm of the signature algorithm (see 5.2.4). The `sig_keyid` parameter in these signatures contains the first six bytes of the associated public role key.
The signature in Fig. 5.8 is created in an Encrypt-than-Mac scheme with the hashing algorithm identified by `sig_hash_algo` and the symmetric key encryption algorithm identified by `sig_enc_algo`(see 5.2.4). The `sig_keyid` parameter in the signature is left empty since the used key is identified through the parameter `ECU\_ID` in the ECU manifest.
In this implementation, `ed25519` was used for all public-key signatures and `AES-256` (as encryption algorithm) together with `SHA3-512` (as hashing algorithm) for the ECU manifest signature.

5.2.3.2 Target Metadata

The target-metadata(-file) for each ECU is 160-bytes in size. The corresponding LIN-Frames are depicted in Fig. 5.5.

`TID` and `TIDv` are described in 4.2.1. Both are chosen to be 8-bytes long in order to support sufficient ECU hardware/software combinations and enough software version numbers per combination. `sa_bytesize` is 6-bytes long, which sets the maximum bytesize of a software image to around 281 terabytes. `sa_compression_algo` defines the algorithm of the chosen compression algorithm for the software-image[6]. `sa_hash_algo` defines the hashing algorithm for `sa_hash` which is the 64-byte software image hash. In this implementation, `SHA3-512` was used for `sa_hash` (see 5.2.4).

[6]Note that software-image compression was not implemented in this proof-of-concept implementation

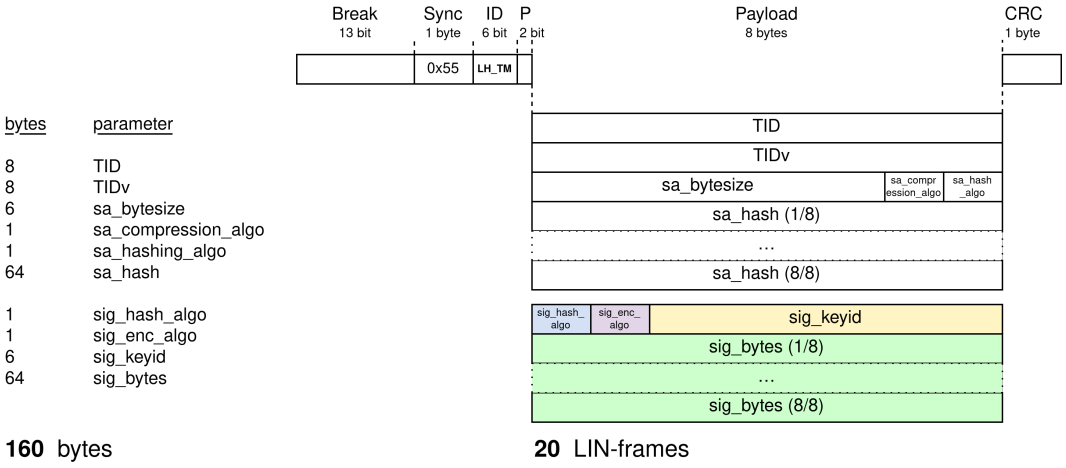

Figure 5.5: LIN Frames of Target-Metadata

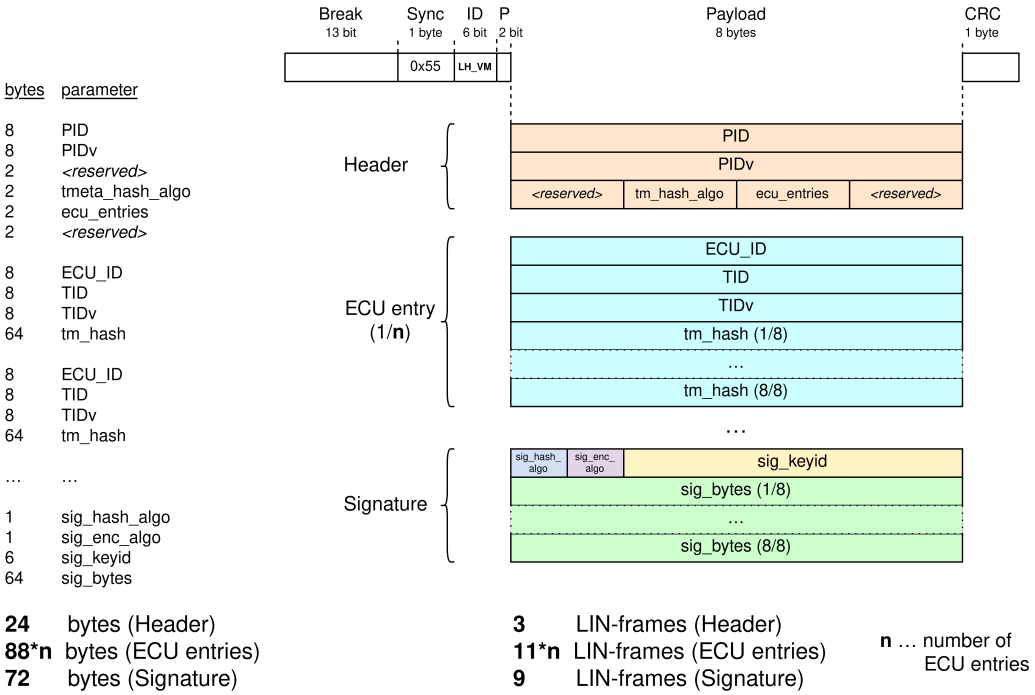

Figure 5.6: LIN Frames of Version-Metadata

5.2.3.3 Version Metadata

As described in 4.3.3.1, version-metadata consists of three parts:
The header (24 bytes), a list of ECU entries (88 bytes per entry), and the signature (72 bytes), see Fig. 5.7.

The header contains `PID` and `PIDv` which were both chosen to be 8-byte long in order to support sufficient bus-system configurations. `tm_hash_algo` and `ecu_entries` are already described in 4.3.3.1 as `hashing algorithm of TM` and `number of ECU entries` respectively. `ecu_entries` is one byte long and defines the number of ECU entries in this version-metadata(-file). Its size limits the maximum number of ECU entries in one version-metadata(-file) to 255 and therefore the maximum size of one version-metadata(-file) to $24 + 88 \cdot 255 + 72 = 22536$ bytes.

Each ECU entry contains the `ECU_ID` of the affiliated ECU, the `TID`, and `TIDv` as well a hash of the associated Target-Metadata. SHA3-512 was used to generate this hash (`tm_hash` in Fig. 5.6). The `ECU_ID` parameter was chosen to be 8-bytes long. As per `TID`, `TIDv`, `PID`, and `PIDv`, this provides around $1.84 \cdot 10^{19}$ ($= 2^{64}$) possible combinations. Even if all OEM's in the world would use the same instance of this update system for 100 years, only $2 \cdot 10^{12}$ distinct ECU_IDs would be generated[7] . The implementation of `TID` and `TIDv` is described in 5.2.3.2.

5.2.3.4 Version Metadata Verification

The version-metadata verification consists of the first 8-bytes of the Version-Metadata-Hash (`vm_hash_id`) and two signatures, one from Target and one from Package, which sets its size to 152 bytes (see Fig. 5.7).

As described in 4.6, ECUs store a version metadata hash within the metadata for each new valid software image. When receiving a VMV, the first 8-bytes of this hash are compared to the received `vm_hash_id` (see Fig. 5.7). Note that the comparison is only performed if the stored hash is not zero. If they

[7]Calculated with 100mio cars per year with 200ECUs each

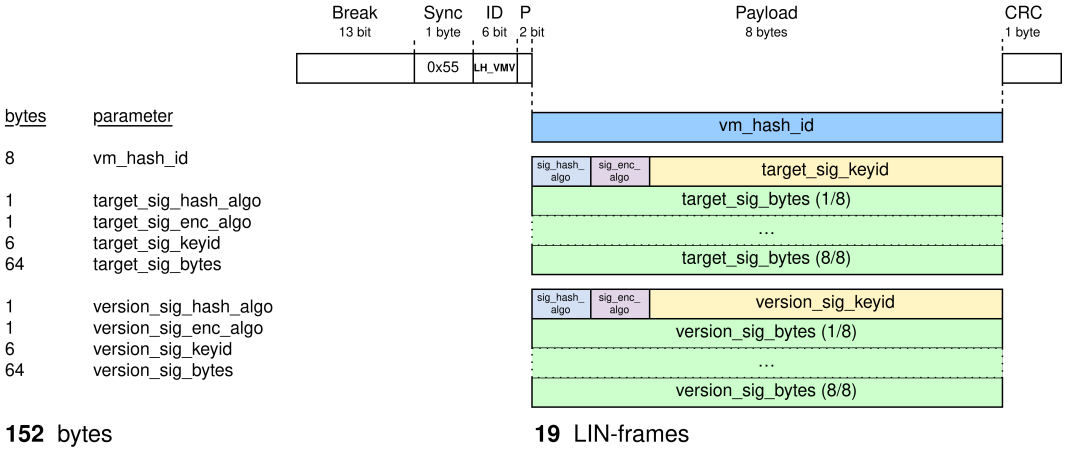

Figure 5.7: LIN Frames of Version-Metadata-Verification

match (and the image is valid and its `TIDv` is adjacent to `currTIDv`), the stored hash in the metadata is set to zero, and `currTIDv` in the bootloader metadata is increased by one to match the value of `TIDv` of this image (making it the new primary image).

Since each ECU calculates and stores the hash of version metadata before VMV is created, the hashing algorithm for the Version-Metadata-Hash is predefined in each bootloader. In this implementation, SHA3-512 was chosen as the hashing algorithm.

5.2.3.5 Software Images

In this implementation, DM creates the software images and stores them in *.bin* files. These binaries are padded to be a multiple of 8-bytes in size. This is done since the payload fields of the LIN-frames during the update are predefined to be 8-bytes long. After an ECU received its Target-Metadata, it expects LIN-Frames with Frame-ID `LIN_SA` containing the bytes of the associated software image. The payload of these frames contains the content of the corresponding binary-file, split into 8-byte chunks. The addressed ECU writes this payload directly to its flash-memory up to `sa_bytesize` in the target metadata (see Fig. 5.5).

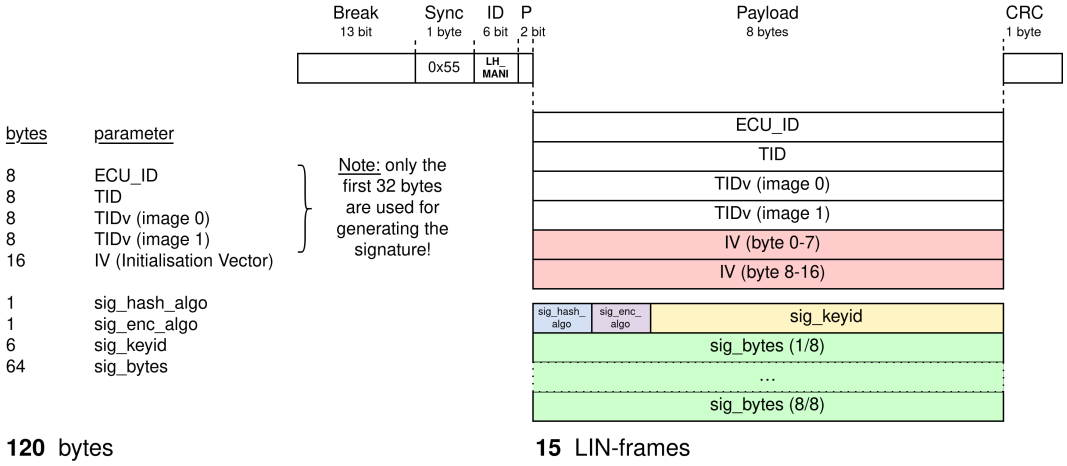

Figure 5.8: LIN Frames of ECU Manifest

5.2.3.6 ECU Manifest

In this implementation, the ECU manifest consists of the 8-byte `ECU_ID` and `TID`, the 8-byte `TIDv` of imageo, and the 8-byte `TIDv` of image1 as well as a 16-byte Initialization-Vector (IV) used for the AES-256 encryption step in the Encrypt-then-Mac signature generation process (see 5.2.3.1).

This IV has to be randomly generated by the ECU for every manifest in order for the encryption process to be secure. For the signature generation, only the first 32-bytes of the ECU Manifest are used (see Fig. 5.8).

5.2.4 Cryptographic Algorithms

In the following sections, the cryptographic algorithms used for this implementation are discussed. Since creating custom and especially secure implementations of these algorithms is out of the scope of this thesis, external sources were used.

5.2.4.1 AES-256

AES-256 is a 128-bit symmetric block cipher algorithm with 14 rounds and a key length of 256-bit. In 2000, AES, or more specifically the underlying algorithm called Rijndael, was selected by the National Institute of Standard and Technology (NIST) to replace 3DES as a symmetric key encryption algorithm. Today, AES is the de-facto standard for symmetric key encryption. Several processing units contain AES hardware accelerators and some even dedicated unprivileged AES instruction sets for very fast computational speed and improved security.
Other than 3DES or AES-128, *AES-256 is totally quantum secure* [Rao et al., 2017]. Grovers algorithm, a quantum algorithm for finding the input of a black-box function for a given output with a high probability, is proven to have the optimal runtime for breaking Rijndael with a best-case runtime of $O(\sqrt{N})$ [Bennett et al., 1997]. The best-case runtime for breaking a 256-bit key with a quantum computer is therefore still 2^{128} iterations, which is considered to be out of reach for any computer available in the next decades.

This property was the reason AES-256 was chosen as an algorithm for the ECUs unique `ECU_AESKEY`. Signatures with `ECU_AESKEY` (e.g. Manifest signature) were computed together with SHA3-512 in an Encrypt-than-Mac scheme. The `tiny-AES-c` implementation[8] was used with Cipher-Block-Chaining (CBC) as a block cipher.

5.2.4.2 EdDSA

The Edwards-curve digital signature algorithm is a digital signature algorithm using twisted Edwards curves instead of Weierstrass curve [Bernstein et al., 2012]. The chosen algorithm variant `ed25519` uses SHA2-512 [Wanzhong et al., 2007] and curve25519 [Bernstein et al., 2012].

EdDSA instantiations such as Ed25519-Original can sign and verify signatures substantially faster than almost all other signatures schemes at similar security

[8] see https://github.com/kokke/tiny-AES-c

levels. For schemes which have comparable speeds, Ed25519-Original further provides considerably smaller signatures, producing 64-byte signatures and 32-byte public keys. Additionally, EdDSA is widely considered to provide better resistance to side-channel attacks than alternative schemes [Brendel et al., 2020].

In this proof-of-concept implementation, `ed25519` is the algorithm used for metadata verification. Regarding the concrete implementation of ed25519, [9] was used. As stated in the readme.txt in the source files, this implementation was extracted from `libsodium` [10] and was not meant to run on an embedded device with limited memory space. Therefore, a drawback of this implementation is its large compiled code size of about 44kB for the GD32V.

5.2.4.3 SHA3-512

As stated in the last section, the used ed25519 implementation already contains a SHA2-512 implementation for its signature algorithm. However, there already exist length extension attacks and preimage attacks on SHA2-512 (see [Dobraunig, Eichlseder, and Mendel, 2015], [Khovratovich, Rechberger, and Savelieva, 2012]).
For future security, its successor SHA3-512 was chosen. `Keccak`, the underlying cryptographic algorithm of SHA3, uses a method called `Sponge-construction` which sets it apart from its predecessors and provides high resistance against the before mentioned attacks.

In this proof-of-concept implementation, SHA3-512 was used for the computation of the hashes of software images and version-/target-metadata (see Fig. 5.3 and Fig. 5.2.3). As source code of the SHA3 hashing algorithm, `SHA3IUF`[11] was used. This implementation comes with an Init-Update-Finalize API, which makes it possible to integrate the software image hash computation directly into the image-to-flash-memory writing process.

[9] see https://github.com/joewalnes/verifysignature
[10] see https://github.com/jedisct1/libsodium
[11] see https://github.com/brainhub/SHA3IUF

5.2.5 Data-Transmission Volume

At the start of the update process, shown in Fig. 4.5, each ECU transmits its ECU Manifest with 120bytes to its Domain Master. Each Domain Master adds the VIN and its own ECU-Manifest to this ECU-Manifest-List, signs it with its own `ECU\_SKEY`, and sends the signed list as Update-Request to VERSION. VINs consists of 17 characters, which makes them slightly larger than the input array of a single SBOX (see 5.2.4).
Padding the VIN to 32 bytes and adding the IV of the AES encryption, adds 48bytes per DM. For five DMs in a vehicle containing 100 ECUs, these Update-Requests would have a sum-total of 12.24kB ($= 100 \cdot 120\text{bytes} + 5 \cdot 48\text{bytes}$).

The maximum possible metadata download volume during an update process in this example for one DM would be approximately 5.7kB.
It consists of a 1856bytes Version-Metadata-File (20 ECU entries), 3200bytes of Target-Metadata (20 Target-Metadata-Files), and a 640byte of Package-Metadata-File (20 ECU entries)[12].

The maximum possible data transfer over the WVI during one update process for a vehicle consisting of 100 ECUs would therefore be 12.24kB of upload traffic and 28.5kB of download traffic (without software images). See 6.2 for more information.

[12]The byte-size of the parameters within the Package-Metadata is based on the implementation of the same parameters in Target- and Version-Metadata

6 EVALUATION

In this chapter, the viability of the proposed update system is discussed. First, its design characteristics are evaluated and compared to existing solutions. Then the properties and characteristic values of the proof-of-concept implementation are compared to benchmark results of existing systems. Finally, possible threats to validity and limitations of the proposed system are considered as well as possible future improvements on the system are discussed.

6.1 Design Evaluation and Comparison

Other than UPTANE and ASSURED, the three metadata-signing roles of BECAUSE do not utilize or expand upon the Update Framework (TUF) directly. However, the same divide-and-conquer principle as in TUF is used by assigning metadata signing tasks to these distinct roles. See section 2.6 for more information on the existing solutions. In the following the BECAUSE framework is compared to the UPTANE and ASSURED framework.

Time-Independence
BECAUSE is time-independent. There are no time-stamp parameters or a dedicated time-stamp role. As Kuppusamy2018 et al. pointed out, *'... ECUs typically do not have real-time clocks'* [Kuppusamy, DeLong, and Cappos, 2018] and therefore have to periodically fetch the current time from an external time-server. To circumvent this problem without making the system susceptible to replay- or freeze-attacks, BECAUSE uses ascending target version numbers (TIDs) and ascending adjacent package version numbers (PIDs).

No Root-Role
A major difference of BECAUSE compared to existing solutions is the lack of a dedicated root role. The root role typically acts as certificate authority by signing, distributing, and revoking the public-keys of all roles in the system. Although the private root key should be stored offline, its compromise breaks the compromise resilience of the whole system since new public-key pairs for each role can be forged. In BECAUSE, INV keeps track of the current public role keys without distributing them. The system, therefore, lacks the possibility to instantly revoke compromised public role keys, but due to the high compromise-resilience, this functionality is not really necessary. Shutting down a compromised role until the problem is fixed should be performed anyway and does prevent new update distributions to be completed (see 4.6 for more information).

Compromise-Resilience without cryptographic requirements on ECU hardware
In the BECAUSE framework, the only secret parameter stored by an ECU is its unique `ECU_SKEY`. If it is compromised, only the corresponding ECU can be compromised. Most low-level ECU lack dedicated secure storage capabilities. Hence, reading `ECU_SKEY` from the flash-memory of an ECU may be possible but requires a physical attack and therefore physical access to the vehicle internals. But for ECUs where the `ECU_SKEY` can be compromised through a physical attack, a physical attack may very well be able to alter flash memory content and therefore lead to arbitrary-software-execution anyway (given that no authenticated boot process exists). Neglecting physical attacks on these low-level ECU which would likely compromise them anyway, an argument can be made that even such low-level ECUs can run BECAUSE fairly secure. This answers **RQ1**, see 3.2.
In order to prevent compromise of the `ECU\_SKEYs` on the INV system, the key storage should be distributed and organized in blackbox-systems for Manifest verification etc.. Another solution would be to use two asymmetric key pairs between the ECU and INV instead of the symmetric ECU_SKEY: the ECU would hold the private key of pair one plus the public key of pair two and INV would hold the public key of pair one and the private key for pair two.

Easy ECU exchange, blacklisting, and decommissioning
A practical advantage arising from the design of the BECAUSE framework is the traceability of distinct ECUs through its `ECU_ID`. INV can verify which ECUs are installed in which vehicles though the ECU-Manifest and thus detect hardware changes in vehicles. INV is, therefore, able to e.g. prevent updates for ECUs those vehicles are reported as stolen or take additional measures. Malicious adversaries are no longer able to strip a stolen vehicle for ECUs and sell them without problems. Decommissioning an ECU at the end-of-life becomes as easy as setting the corresponding database entry on the INV system to inactive or by just dropping it.
In the ASSURED framework, the `Controller` and `Device` are cryptographically bundled by `Controller Key` (see Fig. 2.9) and require a dedicated secure channel. If a `Controller` has to be exchanged in this framework, the keys of all its connected `Devices` in the vehicle have to be updated.
In BECAUSE, only a hardware change request has to be sent to the INV, see section 4.5.3 for more information. No authenticated channel is needed for attestation of proper update installation.

Update system independent from actual software
Due to its dual image setup, the update process has no influence on the already installed image. Hence, an update may fail but can not induce a fault in the current software configuration of the vehicle. Therefore, the update system has no functional impact on the functional safety of the ECU, see 2.5.2. This covers **RQ2**, see 3.2.
Failed updates can be detected through the ECU Manifest and handled according to the information contained in them. Replay attacks with an ECU-Manifest for example can be prevented through storing the hashes of the last received Manifest per ECU, which indicates to INV that the same IV for the symmetric encryption in the Manifest signature creation was used, see 5. Diagnostic functionality and interplay is only defined be by the software image themselves.

6.1.1 Compromise Resilience

In this chapter, the BECAUSE framework is compared to UPTANE and ASSURED in terms of compromise resilience. Eleven relevant attacks on automotive update systems were identified, most of them were previously defined by Kuppusamy et al., see Fig.5 in [Kuppusamy et al., 2016] and by the update framework[1]. The tables presented below show the possible attacks for the most relevant combination of compromised (metadata-signing-)roles with an 'X' symbol. Since all three of the compared update systems feature a role located in the vehicle responsible for update distribution (Primary ECU, Controller, Domain Master), the '**x**' symbol highlights attacks which are only possible if this in-vehicle role is also compromised. All tables consider the attacker having Man-in-the-middle capabilities both inside and outside the vehicle. Physical attacks directly on the MCU of the ECUs are not considered since they are difficult to compare and very implementation-dependent.
A short overview of the considered attacks is given in the following paragraphs.

EA - Eavesdrop attack. An adversary can listen in on the communication.

DoS - Denial-of-Service attack. An adversary can block/jam arbitrary parts of the communication between the entities in the update system.

FEA - Feature-Escalation attack. An adversary manages to install valid software but with premium features enabled. These may be software functions which can be purchased by the OEM to complement the vehicle functionality.

PBIA - Partial-Bundle-Installation attack. This attack prevents some ECUs in an ECU-bundle with overarching functionality (e.g. PID-domain in BECAUSE) from installing the newest software, which in turn may cause compatibility and other issues.

RBA - Rollback attack. This attack *causes an ECU to install outdated software with known vulnerabilities* [Kuppusamy et al., 2016].

[1] see https://theupdateframework.io/security

WIA - Wrong Image attack. This attack causes an ECU to install a valid software image which is not intended for that specific ECU hardware and therefore may lead to exploitable errors in its execution.

MBA - Mixed-Bundle attack. This attack causes ECUs in an ECU-bundle to install software versions which are not compatible with the software versions on other ECUs in the same bundle.

MaMA - Mix-and-Match attack. This attack causes ECUs in the same ECU-bundle to install valid software which is not intended to run on these ECUs at the same time (e.g. incompatible software versions).
Kuppusamy et al. stated that *an attacker is therefore able to arbitrarily combine updates, which is worse than a partial bundle installation or mixed-bundles attack* [Kuppusamy et al., 2016].

ASA - Arbitrary-Software attack. This attack is the worst since it causes ECUs to install and execute arbitrary software.

EDA (Endless-Data-Attack) and **FREEZE** (Freeze-Attack) were intentionally excluded because they are highly implementation-dependent. EDA can be easily prevented by terminating the update process once the received data would exceed the ECUs storage capabilities. FREEZE can be prevented by comparing the current image with the new image (e.g. computing and comparing hashes). BECAUSE is not susceptible to either one of them.
For more information about the attacks see [Kuppusamy et al., 2016].

6.1.1.1 UPTANE

For more detailed information on the compromise resilience of UPTANE, see table 6.1. Even without compromising a single role, UPTANE is vulnerable to PBIA (although the attack can be detected). However, if the director role is compromised, PBIA and even MaMA are not only possible against partial- and full-verification ECUs but also non-detectable.
If a primary ECU is compromised in addition to the director role, partial

verification ECUs are susceptible to almost all listed attacks including ASA. See also Fig. 17 and 18 in [Kuppusamy et al., 2016].

UPTANE

	mild		moderate			major			severe
	EA	DoS	FEA	PBIA	MBA	WIA	RBA	MaMA	ASA
no compromise	X	X		X*					
TR	X	X		X*					
DR	X	X	(**x**)	X		(**x**)	(**x**)	X	(**x**)
TS+RS+DR	X	X	(**x**)	X		(**x**)	(**x**)	X	(**x**)
TS+RS+DR+TR	X	X	X	X		X	X	X	X
RT	X	X	X	X		X	X	X	X

TR... Target Role	X... attack is possible
TS... Timestamp Role	**x**... attack is possible with compromised
RS... Release Role	primary ECU
DR... Director Role	*... attack can be detected
RT... Root Role	( )... attack is possible for partial verification

Table 6.1: Attack Matrix of the UPTANE Framework (Full- and Partial-Verification)

6.1.1.2 ASSURED

For more detailed information on the compromise-resilience of ASSURED, see table 6.2. ASSURED performs slightly better in terms of compromise of multiple roles compared to UPTANE. PBIA is possible with the compromise of the `Domain Controller` role only, which is still better than the zero required roles to be compromised in the UPTANE framework.
If TS and RS are compromised, PBIA, MBA, and MaMA are possible. This

is due to the fact that information about the software version dependencies is only encoded in the metadata signed by two of these roles. As long OEM is not compromised, these three attacks can be detected.
If however OEM and the `Domain Controller` are compromised, every listed attack including ASA is feasible. Since `Domain Controller` verifies all metadata except for the Update Envelope on behalf of the ECU and is also tasked with attesting the state of the ECU, the resulting attacks can not be detected.

ASSURED

	mild		moderate			major			severe
	EA	DoS	FEA	PBIA	MBA	WIA	RBA	MaMA	ASA
no compromise	X	X		**x**					
TS	X	X		**x**					
RS	X	X		**x**					
TR	X	X		**x**					
OEM	X	X	**x**	**x**	**x**	**x**	**x**	**x**	**x**
TS+RS	X	X		X*	X*			X*	
TS+RS+TR	X	X		X*	X*		X*	X*	
RT	X	X		X*	X*		X*	X*	
RT+OEM	X	X	X	X	X	X	X	X	X

TR... Target Role	X... attack is possible
TS... Timestamp Role	**x**... attack is possible with compromised
RS... Release/Snapshot Role	Controller Role
RT... Root Role	*... attack can be detected

Table 6.2: Attack Matrix of the ASSURED framework

6.1.1.3 BECAUSE

For more detailed information on the compromise resilience of the proposed system, see table 6.3. At least three roles need to be compromised in order to launch one of the listed attacks, and there are only two practical combinations:

- If the DM, Target, and Package role are compromised, then it is possible to create a Version-Metadata-Verification for existing Version-Metadata and broadcast it to a PID-domain in which not every ECU has successfully installed the corresponding update. This attack will be detected by INV through the ECU Manifest of the affected ECUs.
- If Target, Version, and Package role are compromised then every attack except MBA can be performed successfully, although detected by INV through the ECU Manifest of the affected ECUs.

If the Version and Package role are compromised, Target prevents execution of new images by not signing new Version-Metadata-Verifications after cross-checking with the INV system.
If the Version and Target role are compromised, Package prevents execution of new images due to the same reason. If the whole INV system is compromised, the ECU_SKEYs are also compromised and proper update installation can no longer be verified. FEA and PBIA are possible, other attacks can be prevented by the knowledge each metadata-signing role possess, and through communication between them. If there is a mechanism in place for renewing ECU_SKEYs, the system can fully recover from such an attack. The compromise of the INV system does not compromise any public keys, see chapter 4.5.4 for more information. As mentioned in chapter 4.3.4, compromising this role should be at least as difficult as compromising Target, Version and Package all together.

Compared to UPTANE and ASSURED, BECAUSE shows a much higher threshold of required roles to be compromised in order for attacks to be feasible.

BECAUSE

	mild		moderate			major			severe
	EA	DoS	FEA	PBIA	MBA	WIA	RBA	MaMA	ASA
no compromise	X	X							
T	X	X							
P	X	X							
V	X	X							
T+P	X	X		**x***					
V+P	X	X							
V+T	X	X							
INV (complete INV system)	X	X	X	X					
T+V+P	X	X	X*	X*		X*	X*	X*	X*
T+V+P+INV	X	X	X	X		X	X	X	X

T... Target Role
P... Package Role
V... Version Role
INV... Inventory Role (complete Inventory system, e.g. TUF instance)

X... attack is possible
x... attack is possible with compromised DM
*... attack can be detected

Table 6.3: Attack Matrix of the BECAUSE framework

6.2 Proof-of-Concept

In the chapter, the proof-of-concept implementation of the proposed system is compared to `TUF` and ASSURED. The metadata sizes as well as the runtime of common operations are compared and discussed below.

6.2.1 Data transmission

In this chapter the size of the transmitted metadata in each update framework is evaluated. **RQ3** is covered in this section, see 3.2. The evaluation is split into two parts: the evaluation of the metadata downloaded by the vehicle (e.g. primary ECU in UPTANE), see Fig. 6.1 and the metadata actually distributed to the individual ECUs, see Fig. 6.2. To make the update process between the systems more comparable, a common configuration for the calculation was chosen:
The update is calculated for one bus-system with one domain controller acting as `primary ECU` for UPTANE, `Controller` for ASSURED, and `Domain Master` for BECAUSE. The update itself consists of one file per ECU. No delegations of metadata signing are performed. Metadata originating from TUF-roles is encoded in the ASN.1 format since it is the most common format used for the distribution of these files in practice. `ed25519` was chosen as the digital signature algorithm for all signatures, except for the root role. Since BECAUSE utilizes SHA512, the hashing algorithms used by the other frameworks were set to be sha512 or at least sha256. Fig. 6.1 depicts the total sum of bytes which has to be fetched by the domain controller from the individual servers. Due to its efficient metadata composition and the missing root-metadata, BECAUSE requires multiple times fewer bytes than UPTANE and ASSURED.

Fig. 6.2 depicts the total sum of bytes which have to be distributed to the individual ECUs on the bus-system. Since ASSURED only sends each ECU its individual update envelope, ASSURED performs slightly better than BECAUSE in this comparison. However, considering automotive bus-systems are unauthenticated channels by default, ASSURED is difficult to

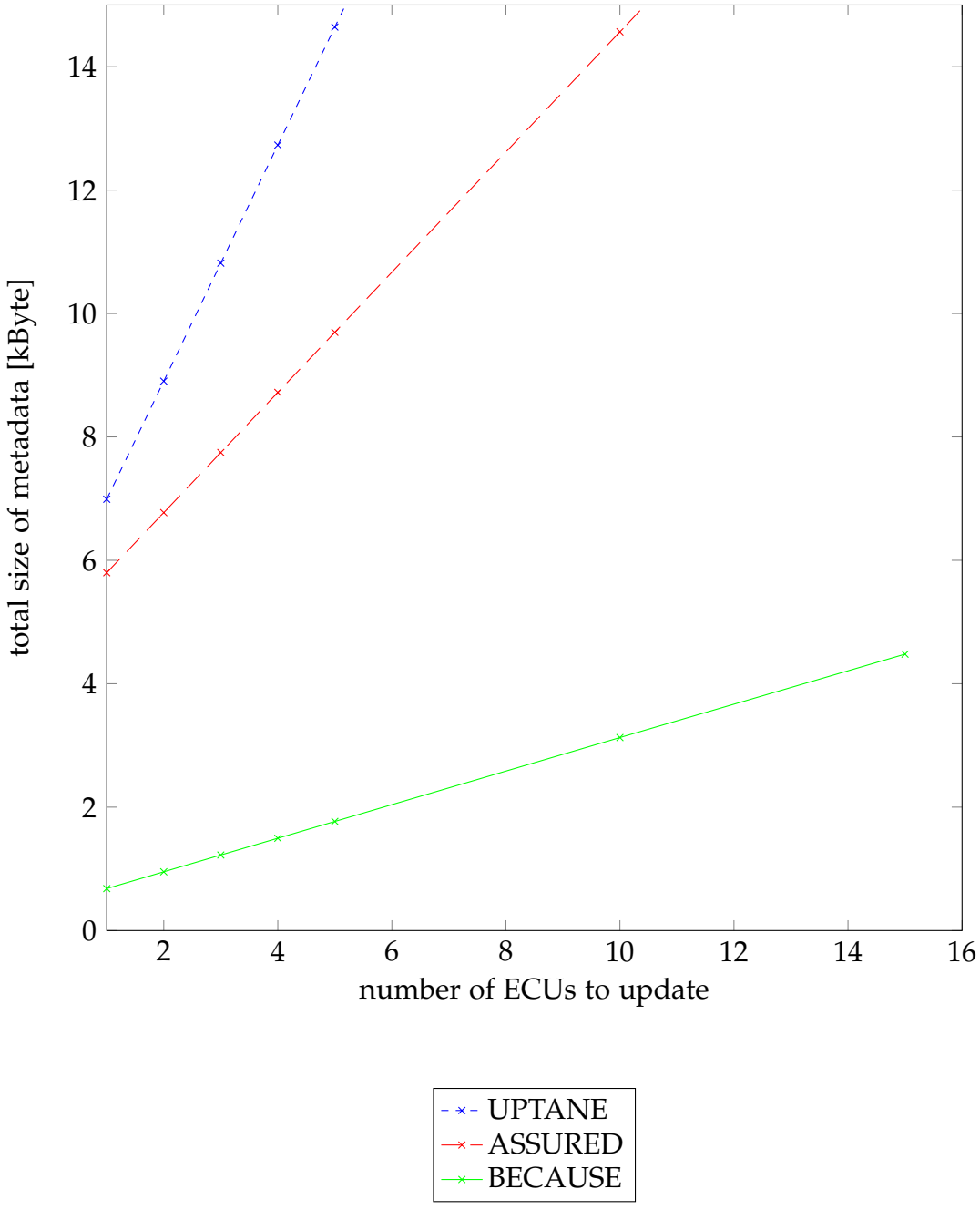

Figure 6.1: Total metadata to be downloaded by the vehicle

compare to the other solutions in this scenario. As in the previous example, the proposed system benefits from its efficient metadata composition. For an update including two ECUs, the complete metadata sent to these two ECUs with the proposed update system is about the same size as one target-metadata file using UPTANE or ASSURED. As shown in chapter 5, BECAUSE can deliver updates over LIN, the slowest bus-system used in modern vehicles. Therefore, any currently used automotive bus-system can be used since bandwidth as the limiting factor is no issue in this case. This answers **RQ3**.

The calculation of the metadata sizes for UPTANE and ASSURED with the given configuration was performed by generating sample metadata according to the example metadata given by the update framework[2]. As proposed by Anton Gerasimov, see [Gerasimov, 2018], the director metadata for UPTANE, was split up into a director-manifest-file and a director-target-file. Both file-sizes were calculated using the structure depicted on page 13 and 14 of [Gerasimov, 2018].
Note that the calculated metadata size for UPTANE and ASSURED may vary depending on the concrete implementation. Fig. 6.1 and Fig. 6.2 should therefore be considered as best-effort depictions of the actual data-traffic for the given configuration.

6.2.2 Runtime Comparison

In this chapter, the measured timings of the most relevant bootloader operations are presented. Timings which include metadata transmission were measured using the proof-of-concept implementation with its 20kBit/s LIN-Bus implementation. All depicted timing values were each calculated from ten distinct measurements.

Table 6.4 depicts the timings of the metadata verification in the different update systems. The values for TUF and ASSURED in table 6.4 were adopted from Table I in Asokan et al., 2018. For more information on the measurements of the metadata verification timings in TUF and ASSURED,

[2] see https://theupdateframework.io/metadata/

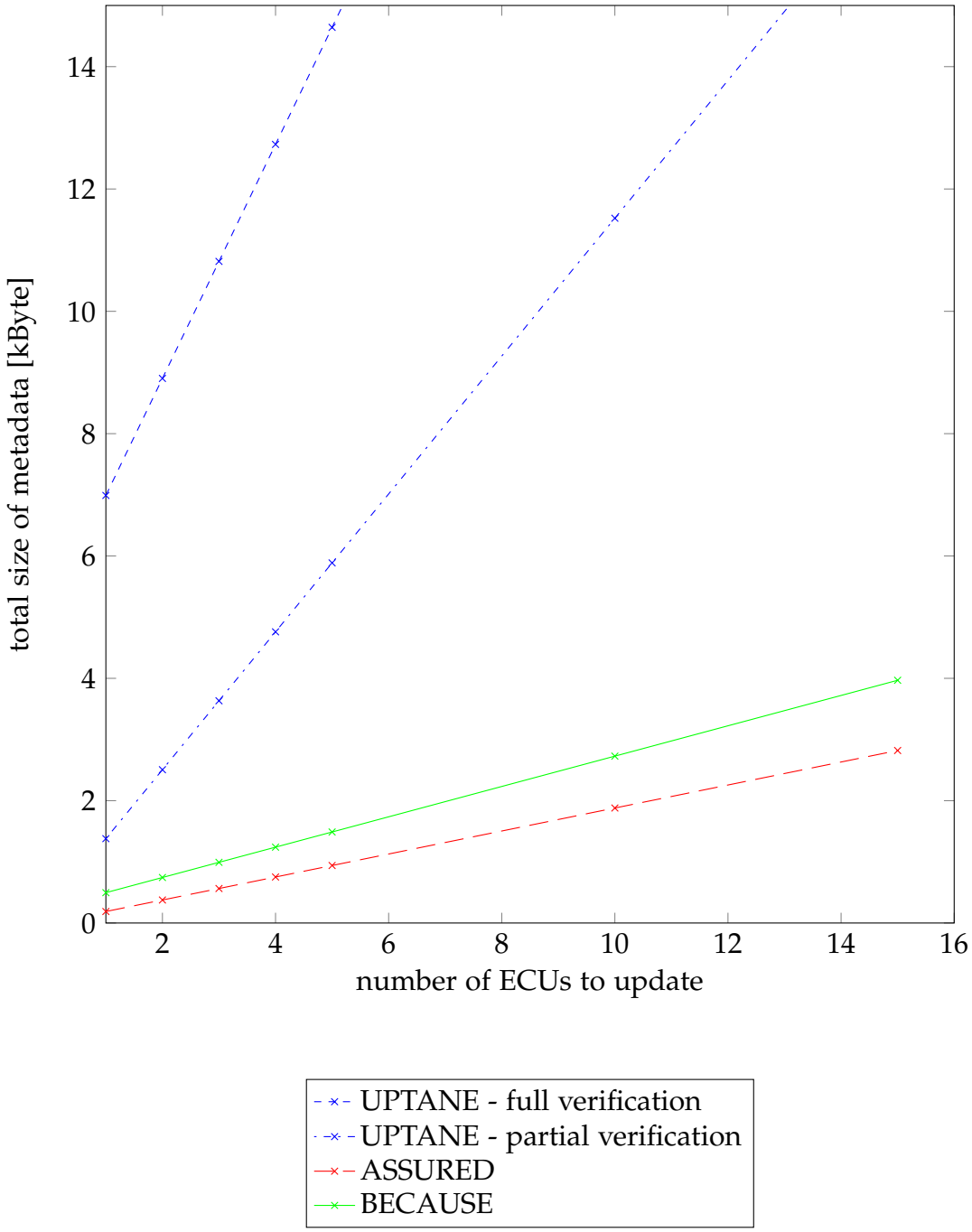

Figure 6.2: Total metadata to be distributed to a set of ECUs on one bus-system

Hardware	I.MX-SABRELITE @ 800MHz		GD32VF103CBT6 @ 108MHz		
System	TUF	ASSURED	BECAUSE Target Metadata	BECAUSE Version Metadata (3 ECU entries)	BECAUSE Total
Time [ms]	14.57	2.56	0.3 ± 0.1	0.5 ± 0.1	0.8 ± 0.2

Table 6.4: Metadata Verification Time (source of data for TUF and ASSURED: Asokan et al., 2018, Table I)

see [Asokan et al., 2018]. Since these timings were measured on different hardware with different clock speeds, the values can not be directly compared. However, since all systems use `ed25519` for signature verification and assuming the verification times are mainly dependent on the number of signatures to verify in combination with the processing units clock-speeds, BECAUSE performs several times better than both TUF and ASSURED.

	Bootloader Verification	create and send ECU Manifest	transmit, receive and verify metadata: Version Metadata (3 ECU entries)	transmit, receive and verify metadata: Target Metadata
Time [ms]	18 ± 1	199 ± 3	227 ± 5	123 ± 1

Table 6.5: Measured timings for bootloader operations (20kBit/s LIN-bus system)

Table 6.5 depicts the timings of the most relevant operations of the proposed system. These timings are each the calculated mean values of 10 consecutive measurements taken with the proof-of-concept. The results are discussed in the following. The bootloader verification, where the bootloader verifies the integrity of the stored metadata and its images at the start of its execution, takes about 18ms. This is mainly due to the SHA3-512 hash computations. The time between the reception of the ECU Manifest request, the creation of the ECU Manifest and its complete transmission takes around 200ms. The time between the start of Version-Metadata transmission by the DM and its verification being completed by the ECU takes around 230ms and 123ms for the Target-Metadata. The variations in the timings are due to the LIN-Bus communication.

All bootloader operations combined, except the reception of a software

image, can be performed in under one second. The execution time for an update is therefore almost exclusively dependent on the size of the software image(s) and the speed of the bus-system between the DM and the ECU. Through modifying the LIN-bus protocol for the communication between the DM and ECU bootloaders, by shortening the Break-Field of the LIN-Frames carrying the bytes of the software image, a complete 30kB update for one ECU (without VMV), takes only around 20 ± 1 seconds.

6.3 Limitations and Future Work

LIMITATION: Memory Consumption
The main drawback of this solution is the flash-memory consumption, mainly caused by the two images being stored at the same time. However, this 2-image approach provides significantly higher fault-tolerance than e.g. delta-update approaches and is therefore an integral part of update systems like UPTANE, see [Community, 2019] for more information.
As the proof-of-concept implementation and also ASSURED have shown, more than 50% of the bootloader size is used for the implementation of the verification algorithm, namely `ed25519` in both cases. For further information see 5.2.4. In contrast, all metadata stored by an ECU combined only accounts for less than 1kB in the proof-of-concept implementation, see 5.3. This also answers **RQ4**, see 3.2.

LIMITATION: Authenticated ECU replacement
Every ECU is identified by a unique `ECU_ID` which is assigned to a specific PID-domain within a vehicle, which in turn is identified by a unique VIN. Information about this relation is encoded in every type of metadata. This approach allows for easy ECU blacklisting and decommissioning, see 6.1, but also requires an update of the associated INV database entry for every ECU exchange. The consequence is the requirement for an authenticated ECU exchange system. An example is given in 4.5.3. The OEM is therefore able to limit third-party workshops from exchanging ECUs and favor its own workshops. This trend can already be seen with manufacturer specific diagnostics, see 2.4.

FUTURE WORK: Updateable Crypto-modules
As described in 3, vehicles today have an expected operation phase of up to fifteen years. For the proof-of-concept implementation, cryptographic algorithms were chosen which are expected to provide sufficient resistance against computational capabilities available in this time period. Nevertheless, flaws in these cryptographic principles are likely to be discovered during this period. It may be advantageous to organize the cryptographic algorithms for the bootloader into modules and provide means for updating/exchanging them. BECAUSE is already designed with this future improvement in mind by explicitly defining metadata fields that identify the used cryptographic primitives (e.g. hashing algorithms).

FUTURE WORK: PID-Change procedure
As mentioned in 4.6, a process for a change of the PID of a PID-domain was not implemented but is supported by the design of the proposed system and may hold several advantages. An OEM can already predefine several PIDs (with different TIDs) for the same hardware. An update of a PID identifier would allow a customer to purchase premium functionality like more detailed road-maps or in-vehicle internet access any time after the vehicle purchase. This would also allow a manufacturer to sell the same vehicle model to every customer and upgrade the PID values according to the purchased features during the first update.

FUTURE WORK: Compatibility between manufacturers
An ECU from one supplier may be used in many vehicles from several automotive manufacturers. It may be advantageous for all parties using the proposed system, to provide an overarching system for assigning TID/TIDv numbers. Otherwise, the same ECU with the same functionality will have different ECU_IDs in the manufacturer's ecosystems. Regarding diagnostic functionality, this approach can become rather complex. However, compatibility between different manufacturers/automotive brands would probably provide more advantages than disadvantages.

7 CONCLUSION

In this thesis, a new automotive update system was proposed, which adopts the principle of multiple roles signing different metadata. The resulting roles are more specifically tailored to accustom the requirements of the automotive industry than previous update systems. This results in higher compromise resilience against the most relevant attacks on automotive update systems.
The compact metadata sizes allow for fast update delivery even on the lowest-speed bus-systems in state-of-the-art vehicles and also reduces the flash-memory required for its storage. Procedures for initial ECU coding/flashing at the end of the production line, the replacement of faulty ECUs as well their decommissioning are included in the design of the proposed solution. Its non-dependability from the actual software functionality of the ECU, the diagnostic specification of the overall system as well as secure storage capabilities of the ECU makes its integration in existing systems rather trivial.
The proof-of-concept implementation shows that the proposed system is practicable and more efficient than current update systems in terms of metadata overhead and therefore update distribution.

Appendix

BIBLIOGRAPHY

Al-Ashaab, A. et al. (2014). "Set-based concurrent engineering model for automotive electronic/software systems development." In: *Competitive Design - Proceedings of the 19th CIRP Design Conference*. January 2009, pp. 464–468. URL: https://www.researchgate.net/publication/228901184 (cit. on p. 3).

Al-Taee, Majid A., Omar B. Khader, and Nabeel A. Al-Saber (2007). "Remote monitoring of vehicle diagnostics and location using a smart box with global positioning system and general packet radio service." In: *2007 IEEE/ACS International Conference on Computer Systems and Applications, AICCSA 2007*, pp. 385–388. DOI: 10.1109/AICCSA.2007.370910 (cit. on p. 5).

Asokan, N. et al. (2018). "ASSURED: Architecture for secure software update of realistic embedded devices." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11, pp. 2290–2300. ISSN: 02780070. DOI: 10.1109/TCAD.2018.2858422. arXiv: 1807.05002 (cit. on pp. 32, 34–36, 39, 52, 96, 98).

Barber, Angela (2018). "Status of Work in Process on ISO/SAE 21434 Automotive Cybersecurity Standard." In: pp. 1–25. URL: https://fdocuments.net/document/isosae-21434-cybersecurity-engineering-proposal-isosae-21434-purpose-the.html (cit. on p. 28).

Bennett, Charles H. et al. (1997). "Strengths and weaknesses of quantum computing." In: *SIAM Journal on Computing* 26.5, pp. 1510–1523. ISSN: 00975397. DOI: 10.1137/S0097539796300933. arXiv: 9701001 [quant-ph] (cit. on p. 82).

Bernstein, Daniel J. et al. (2012). "High-speed high-security signatures." In: *Journal of Cryptographic Engineering* 2.2, pp. 77–89. ISSN: 21908508. DOI: 10.1007/s13389-012-0027-1 (cit. on p. 82).

Bosch (2012). "CAN with Flexible Data-Rate Specification Version 1.0." In: p. 32. URL: https://can-newsletter.org/assets/files/ttmedia/raw/e5740b7b5781b8960f55efcc2b93edf8.pdf (cit. on p. 20).

Brendel, Jacqueline et al. (2020). "The Provable Security of Ed25519: Theory and Practice." In: *IACR Cryptology ePrint Archive* Report 2020/823, pp. 1–25. URL: https://eprint.iacr.org/2020/823 (cit. on p. 83).

Broadcom, Cooperation (2014). "BroadR-Reach® Physical Layer Transceiver Specification For Automotive Applications." In: URL: http://www.ieee802.org/3/1TPCESG/public/BroadR_Reach_Automotive_Spec_V3.0.pdf (cit. on p. 22).

Chakraborty, Sandip and Sukumar Nandi (2013). "IEEE 802.11s mesh backbone for vehicular communication: Fairness and throughput." In: *IEEE Transactions on Vehicular Technology* 62.5, pp. 2193–2203. ISSN: 00189545. DOI: 10.1109/TVT.2013.2239672 (cit. on p. 4).

Charette, Robert N (2009). "This car runs on code." In: *IEEE Spectrum* 46.3, p. 3. URL: https://spectrum.ieee.org/transportation/systems/this-car-runs-on-code (cit. on p. 9).

Chowdhury, Thomas et al. (2018). "Safe and Secure Automotive Over-the-Air Updates." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 11093 LNCS, pp. 172–187. ISSN: 16113349. DOI: 10.1007/978-3-319-99130-6_12 (cit. on p. 26).

Community, Uptane Alliance (2019). "Uptane IEEE-ISTO Standard for Design and Implementation." In: URL: https://uptane.github.io/papers/ieee-isto-6100.1.0.0.uptane-standard.html (cit. on p. 99).

Dobraunig, Christoph, Maria Eichlseder, and Florian Mendel (2015). "Security Evaluation of SHA-224, SHA-512/224, and SHA-512/256." In: February, pp. 1–44. URL: https://www.cryptrec.go.jp/exreport/cryptrec-ex-2401-2014.pdf (cit. on p. 83).

Economic Commission for Europe of the United Nations (2019). "Regulation No 83 of the Economic Commission for Europe of the United Nations (UNECE) — Uniform provisions concerning the approval of vehicles with regard to the emission of pollutants according to engine fuel requirements [2019/253]." In: *Official Journal of the European Union*. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=OJ:L:2019:045:FULL&from=EN (cit. on p. 24).

Farrugia, Mario et al. (2017). "The usefulness of diesel vehicle onboard diagnostics (OBD) information." In: *Proceedings of the 2016 17th International Conference on Mechatronics - Mechatronika, ME 2016* (cit. on p. 24).

Ford Motor Company (1919). *Ford Model-T Manual*. URL: https://archive.org/details/1919_Ford_Model-T_Manual (cit. on p. 2).

Gai, Paolo and Massimo Violante (2016). "Automotive embedded software architecture in the multi-core age." In: *Proceedings of the European Test Workshop* 2016-July. ISSN: 15581780. DOI: 10.1109/ETS.2016.7519309 (cit. on p. 13).

Gerasimov, Anton (2018). "Secure OTA updates for small devices with Uptane and RIOT." In: *RIOT Summit*. HERE Technologies. URL: http://summit.riot-os.org/2018/wp-content/uploads/sites/10/2018/09/3_4-Anton-Gerasimov-OTA.pdf (cit. on p. 96).

Halder, Subir, Amrita Ghosal, and Mauro Conti (2019). "Secure OTA Software Updates in Connected Vehicles: A survey." In: 26262, pp. 1–18. arXiv: 1904.00685. URL: http://arxiv.org/abs/1904.00685 (cit. on pp. 7, 9, 37).

Hank, Peter et al. (2013). "Automotive ethernet: In-vehicle networking and smart mobility." In: *Proceedings -Design, Automation and Test in Europe, DATE*, pp. 1735–1739. ISSN: 15301591. DOI: 10.7873/date.2013.349 (cit. on p. 23).

International Organization for Standardization (2003). "ISO 11898-1." In: 2003, p. 52. URL: http://read.pudn.com/downloads209/ebook/986064/ISO11898/ISO11898-1.pdf (cit. on p. 18).

Kamkar, Samy (2015). *Drive It Like You Hacked It*. URL: https://samy.pl/defcon2015/2015-defcon.pdf (cit. on p. 7).

Khovratovich, Dmitry, Christian Rechberger, and Alexandra Savelieva (2012). "Bicliques for preimages: Attacks on Skein-512 and the SHA-2 family." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7549 LNCS, pp. 244–263. ISSN: 03029743. DOI: 10.1007/978-3-642-34047-5_15 (cit. on p. 83).

Klanner, Wilfried et al. (2004). *Unfallverletzungen in Fahrzeugen mit Airbag*. ISBN: 3865091938 (cit. on p. 9).

Kuppusamy, Trishank Karthik, Lois Anne DeLong, and Justin Cappos (2018). "Uptane: Security and Customizability of Software Updates for Ve-

hicles." In: *IEEE Vehicular Technology Magazine* 13.1, pp. 66–73. ISSN: 15566072. DOI: 10.1109/MVT.2017.2778751 (cit. on p. 85).

Kuppusamy, Trishank Karthik et al. (2016). *Uptane: Securing Software Updates for Automobiles*. URL: https://uptane.github.io/papers/kuppusamy_escar_16.pdf (cit. on pp. 31, 32, 34, 40, 52, 88–90).

Mercedes-Benz Canada, Inc. (2005). *Instrument Cluster SCN Coding for Component Replacement*. URL: http://www.cardiagnostics.be/-now/DAS-WIS_bestanden/Instrument_Cluste_SCN_Coding.pdf (cit. on p. 17).

Miller, Charlie and Chris Valasek (2015). "Remote Exploitation of an Unaltered Passenger Vehicle." In: *Defcon 23* 2015, pp. 1–91. URL: http://illmatics.com/RemoteCarHacking.pdf (cit. on pp. 6, 37).

Mössinger, Jürgen (2010). "Software in automotive systems." In: *IEEE Software* 27.2, pp. 92–94. ISSN: 07407459. DOI: 10.1109/MS.2010.55 (cit. on p. 10).

Nakamoto, Yuya, Daisuke Nishijima, and Shigemi Kagawa (2019). "The role of vehicle lifetime extensions of countries on global CO2 emissions." In: *Journal of Cleaner Production* 207, pp. 1040–1046. ISSN: 09596526. DOI: 10.1016/j.jclepro.2018.10.054. URL: https://doi.org/10.1016/j.jclepro.2018.10.054 (cit. on p. 6).

OSEK/VDX (2005). "OSEK/VDX history and structure." In: *IEE Colloquium (Digest)* 523. ISSN: 09633308. DOI: 10.1049/ic:19981073 (cit. on p. 30).

Rao, Sandeep Kumar et al. (2017). "The AES-256 Cryptosystem Resists Quantum Attacks." In: *International Journal of Advanced Research in Computer Science* 8.3, pp. 404–408. ISSN: 0976-5697 (cit. on p. 82).

Ribeiro, Leandro Batista and Marcel Baunach (2017). "Towards dynamically composed real-time embedded systems." In: *Informatik aktuell book series*. Springer Fachmedien, pp. 11–20. DOI: 10.1007/978-3-662-55785-3_2. URL: https://link.springer.com/chapter/10.1007/978-3-662-55785-3_2 (cit. on p. 11).

Satnam Singh, Bangalore (IN); Vineet R. Khare, Bangalore (IN); Rahul Chougule, Bangalore (IN) (2013). *FAULT DIAGNOSIS AND PROGNOSIS USING DAGNOSTIC TROUBLE CODEMARKOV CHAINS*. URL: https://patentimages.storage.googleapis.com/c8/de/e0/aaf56115a69026/US8498776.pdf (cit. on p. 25).

Schäuffele, Jörg and Thomas Zurawka (2003). *Automotive software engineering*. 6th ed. Springer Fachmedien, pp. 719–720. ISBN: 9783658118143. DOI: 10.1007/1-4020-7991-5_21 (cit. on pp. 12, 17, 37).

Schmittner, Christoph (2019). "Automotive Cybersecurity and SW Updates." In: *FUSACOM*. AIT (cit. on pp. 6, 29, 39).

Schmittner, Christoph and Georg Macher (2019). "Automotive Cybersecurity Standards - Relation and Overview." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 11699 LNCS, pp. 153–165. ISSN: 16113349. DOI: 10.1007/978-3-030-26250-1_12 (cit. on pp. 26, 27).

Shanmugam, Karthik (2014). "Secure Software Update Mechanism for Automotive ECU." In: *International Journal of Innovative Research in Advanced Engineering (IJIRAE)* 1.10, pp. 246–249 (cit. on p. 1).

Skavhaug, Amund et al. (2016). "Computer Safety, Reliability, and Security." In: *SAFECOMP*. ISBN: 9783319454801 and 3319454803. DOI: 10.1007/978-3-319-45480-1. URL: http://dx.doi.org/10.1007/978-3-319-45480-1 (cit. on p. 28).

Steger, Marco et al. (2016). "Generic framework enabling secure and efficient automotive wireless SW updates." In: *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA* 2016-Novem. ISSN: 19460759. DOI: 10.1109/ETFA.2016.7733575 (cit. on p. 4).

Steger, Marco et al. (2018). "Secure Wireless Automotive Software Updates Using Blockchains: A Proof of Concept." In: pp. 137–149. DOI: 10.1007/978-3-319-66972-4_12 (cit. on p. 1).

Susumu Akiyama, Kariya (JP) (2015). *VEHICULAR RELAY DEVICE, IN-VEHICLE COMMUNICATION SYSTEM, FAILURE DIAGNOSTIC SYSTEM, VEHICLE MANAGEMENT DEVICE, SERVER DEVICE AND DETECTION AND DIAGNOSTIC PROGRAM*. URL: https://patentimages.storage.googleapis.com/e3/12/65/fadfa7db2f2026/US6694235B2.pdf (cit. on p. 14).

Technology, Globalplatform Device (2016). *TEE Management Framework v1.0*. URL: https://globalplatform.org/wp-content/uploads/2018/06/GPD_TEE_MgmtFramework_v1.0_PublicRelease.pdf (cit. on p. 35).

Upstream Security Inc. *AutoThreat Intelligence Cyber Incident Repository*. URL: https://www.upstream.auto/research/automotive-cybersecurity/ (visited on 11/25/2020) (cit. on p. 1).

Wanzhong, Sun et al. (2007). "Design and optimized implementation of the SHA-2(256, 384, 512) hash algorithms." In: *ASICON 2007 - 2007 7th International Conference on ASIC Proceeding* 2.3, pp. 858–861. DOI: 10.1109/ICASIC.2007.4415766 (cit. on p. 82).

Wouters, Lennert et al. (2019). "Fast, Furious and Insecure: Passive Keyless Entry and Start Systems in Modern Supercars." In: *tCHES 2019* 2019, Issu.3, pp. 66–85. DOI: 10.13154/tches.v2019.i3.66-85. URL: https://tches.iacr.org/index.php/TCHES/article/view/8289 (cit. on p. 7).

Yang, Yalian et al. (2013). "Research and development of hybrid electric vehicles can-bus data monitor and diagnostic system through OBD-II and android-based smartphones." In: *Advances in Mechanical Engineering* 2013. ISSN: 16878132. DOI: 10.1155/2013/741240 (cit. on p. 24).