Centre for Electromobility Singapore

Technische Universität München

MASTER THESIS

DESIGN AND IMPLEMENTATION OF A SIMULATOR FOR TIMING ANALYSES OF AUTOMOTIVE NETWORKS

ARTUR MROWCA

TUM CREATE LTD., SINGAPORE

INSTITUTE FOR REAL-TIME COMPUTER SYSTEMS, TECHNISCHE UNIVERSITÄT MÜNCHEN

October 26, 2015

Design and Implementation of a Simulator for Timing Analyses of Automotive Networks

Master-Thesis

Supervised by Lehrstuhl für Realzeit-Computersysteme
Technische Universität München
Prof. Dr. sc. Samarjit Chakraborty

Executed at TUM CREATE Ltd., Singapore

Advisor: Dipl.-Ing. (TUM) Philipp Mundhenk

Author: Artur Mrowca

Submitted in October 2015

Author's Declaration

Herewith I confirm that I independently prepared this thesis. No further references or auxiliary means as the ones indicated in this work were used for the preparation.

Singapore, October 26, 2015

. . . . . .
Artur Mrowca

Abstract

Security in internal automotive networks is a rising topic in today's research as those networks offer increasingly more interfaces to external networks that can be exploited by attackers in order to steal, sabotage or manipulate cars. Those attacks are possible, as in modern automobiles there are no security mechanisms implemented that manage the access to message flows between Electronic Control Units (ECUs). In order to solve this problem, authentication protocols, as they are already common practice in computer networks, need to be implemented in those networks. In safety critical parts of those networks different circumstances are given as in computer networks, including lower computational capacity and the necessity for real-time behavior. Hence, in order to analyze to which extent existent or proposed authentication protocols fulfill the requirements of automotive networks, simulation is used as a cost effective tool.

Comparable simulators mostly focus on the analysis of computer networks or on the validation of intra-vehicular networks. However, none is focused on the validation and analysis of security protocols in intra-vehicular networks. That is why, in this thesis a discrete-event simulator is presented that fills this gap.
The proposed simulator provides mechanisms to implement the timing behavior of security mechanisms based on realistic time values. The temporal behavior can be parametrized via project parameters, such as chosen algorithms that are used within the authentication protocols. Realistic timing values and their dependencies to project parameters are determined via measurements on a STM32 microcontroller. Additionally, the simulator is designed in a modular way, as it enables the creation of variable components in an user-defined intra-vehicular network architecture. This modularity is enhanced by interchangeable modules in each component, that define their communication behavior on each layer of the OSI reference model. Also, it provides interfaces to define user-defined modules and provides analysis and access tools.
This allows to validate authentication protocols on a temporal and conceptual basis under different configurations. Also it allows to compare protocols and to determine their applicability in internal automotive networks. In this thesis, the simulator was used to implement a CAN network with the Transport Layer Security (TLS), Timed Efficient Stream Loss-Tolerant Authentication (TESLA) and Lightweight Authentication for Secure Automotive Networks (LASAN) protocols. It was shown that it is possible to analyze architectures with over 1000 messages via 100 ECUs.

Acknowledgements

First of all, I would like to express my deepest sense of Gratitude to my supervisor Philipp Mundhenk. His systematic guidance, expertise, continuous support and great effort made this thesis and its outcome possible in the way it is presented here.

My appreciation also goes to Dr. Sebastian Steinhorst for his ideas, valuable remarks and great expertise throughout my time at TUM CREATE Ltd..
Also I would like to thank Dr. Martin Lukasiewycz and Samantha Lee together with the HR department of TUM CREATE Ltd. for helping me with all administrative questions. Thanks goes also to them and Prof. Dr. sc. Smarjit Chakraborty for making it possible for me to do this Master Thesis at TUM CREATE Ltd. in Singapore.
A special thanks goes to Arne Meeuw, who supported me with his great expertise about battery management systems, fruitful discussions and encouragement.
Furthermore, I like to thank the whole team of RP3 (embedded systems) at TUM CREATE Ltd. for their insightful comments and the chance to get an insight into their research fields.

I am thankful to my colleagues Michael Popow, Marcel Schmitt, Florian Surek, Maximilian Storp, Sebastian Ghanbari, Thomas Baumeister and all other students at TUM CREATE for the great team spirit, encouragement and joy during my time at TUM CREATE Ltd. in Singapore.
Lastly, thanks also go to my family and friends, as well as my partner Veronika Barthuber for their unwavering faith and support throughout the time of this thesis.

Contents

List of Figures

Glossary

ACK Acknowledgement

ADAS Advanced Driver Assistance Systems

AES Advanced Encryption Standard

API Application Programming Interface

AVB Audio Video Bridging

BMS Battery Management System

BRS Bit Rate Switch

CA Certificate Authority

CAN Controller Area Network

CAN FD Controller Area Network with Flexible Data-Rate

CAs Certificate Authorities

CMU Cell Management Unit

CPCSF Cyber-Physical Co-Simulation Framework

CRC Cyclic Redundancy Check

CSMA Carrier Sense Multiple Access

CSV Comma-Separated Value

DES Data Encryption Standard

ECC Elliptic Curve Cryptography

ECU Electronic Control Unit

EDL Extended Data Length

ESI Error State Indicator

GPS Global Positioning System

GUI Graphical User Interface

HiL Hardware-in-the-Loop

HTTPS Hypertext Transfer Protocol Secure

IFS Inter Frame Space

ISO International Organization for Standardization

IVNS Intra-Vehicular Network Simulator

LASAN Lightweight Authentication for Secure Automotive Networks

LIN Local Interconnect Network

LVDS Low Voltage Differential Signaling

MAC Message Authentication Code

MAD Medium Average Distribution

MD5 Message-Digest-Algorithm 5

MiL Model-in-the-Loop

MIPS Million Instructions Per Second

MOST Media Oriented Systems Transport

MVC Model-View-Controller

NRZ Non-Return-to-Zero

OBD On-Board-Diagnostics

OSI Open Systems Interconnection

PCI Protocol Control Information

PiL Processor-in-the-Loop

PRF Pseudo Random Function

PROM programmable read-only memory

RSA Rivest-Shamir-Adleman

RTR Remote Transmission Request

SAE Society of Automotive Engineers

SBM Sensor and Balancing Module

SHA1 Secure-Hash-Algorithm 1

SHA256 Secure-Hash-Algorithm 256

SiL Software-in-the-Loop

SM Security Module

SoC State of Charge

SSL Secure Sockets Layer

STMCL STM Cryptographic Library

TA Trusted Authority

TDMA Time Division Multiple Access

TESLA Timed Efficient Stream Loss-Tolerant Authentication

TLS Transport Layer Security

TPMS Tire Pressure Monitoring System

USB Universal-Serial-Bus

V2I Vehicle-to-Infrastructure

V2V Vehicle-to-Vehicle

1 Introduction

Since the first patent of the automobile in 1886 by Carl Benz [Sac92] until today, the purpose and principle of the car remained the same. Transportation by moving a motorized chassis on wheels controlled by a steering wheel and brakes. While on the outside this picture did not change over the years, on the inside multiple inventions transformed the car from a purely mechanical device to a complex electromechanical system. To ensure safety, pleasure and comfort in automobiles, ECUs were introduced that are interconnected in an intra-vehicular network to perform control and sensing tasks in the car. While this internal network of ECUs was closed to other networks (e.g. Internet) for many years, in the last decade interfaces such as Bluetooth, Wi-Fi or 3G/4G were added to those networks. This makes those intra-vehicular networks accessible to the outside world. While, this increases the safety and the comfort of the car, with systems such as Vehicle-to-Vehicle (V2V) or Vehicle-to-Infrastructure (V2I), it also awakens the urge for security mechanisms in those communication systems.
One solution to implement security are authentication protocols which are already established in computer networks. Those protocols function well in those networks. However, it is not proven if they fulfill the temporal and conceptual requirements of internal automotive networks. That is why, in this thesis a discrete-event simulator is proposed that enables the analysis of those authentication protocols in internal automotive networks.
In this chapter in Section 1.1 the motivation for the simulator is presented and in Section 1.2 the contribution of this simulator is described. Lastly, in Section 1.3 an outline is given.

1.1 Motivation

Security in internal automotive networks is becoming more and more important. This is due to the fact that those networks have increasingly more interfaces to external networks that can be

exploited by attackers in order to steal, sabotage or manipulate cars [WWW07]. Those interfaces include mechanisms to improve the safety of the car, such as General Motor's "OnStar" technology, that is used to send automatic crash reports in case of an accident or to recover stolen cars [KCR+10]. Other interfaces that can be exploited are the Tire Pressure Monitoring System (TPMS) that is built in in most modern cars [RMM+10] or interfaces for entertainment, such as Bluetooth or Wi-Fi, that are used to pair the mobile phone with the infotainment system [SNA+13b]. In previous works, it was shown that those external connections can be used to eavesdrop and reverse engineer messages [RMM+10] or to send messages on the bus that can directly influence safety critical components such as the steering or the brakes [CMK+11].
Those attacks are possible as in modern automobiles there are no security mechanisms implemented that manage the access to message flows between ECUs [SNA+13b]. Thus, for instance in [RMM+10] attackers were able to eavesdrop messages that are sent within the TPMS as its communication is unencrypted. This enabled them to reverse engineer the protocol that is used in this system using GNU Radio and the Universal Software Radio Peripheral (USRP). Consequently, they were able to trigger TPMS messages in the car via a radio station. This is why, in [MSL+15] or [GMVV12] authentication protocols are proposed to solve this problem. Those protocols exchange encryption keys between senders and receivers of a communication. This enables the communicating partners to exchange encrypted messages that can only be read by authentic nodes that posses the according key.
While authentication protocols are already common practice in computer networks, it is not yet validated to which extent they fulfill the requirements of automotive networks. This is because those protocol require cryptographic operations that are computationally expensive and the computational capacity of state-of-the-art ECUs is limited. On top of that, in safety critical parts of those networks real-time behavior is required. Thus, as authentication protocol add latencies to each transmitted message, it has to be proved that timing constraints are still fulfilled when those protocols are used. Also, in safety critical internal automotive networks, ECUs transmit messages as broadcasts. This allows attackers to connect to the network and to manipulate or eavesdrop messages that are sent, which is a harm to the privacy of the driver. Thus, in order to counteract this, authentication protocols in intra-vehicular networks need to enable encrypted and secure multicast connections between the ECUs. This can be done by providing keys to senders and receivers of each message stream. This key can then be used to perform encrypted multicast communication.
In addition to that, from a car manufacturer's perspective, it is of high importance to decide at an early development stage, which authentication protocol is suited best to be implemented in intra-vehicular networks of the next generation of cars. This saves costs both in research and development. That is why the suitability of authentication protocols in an internal automotive network has to be determined.
This can be done by sequentially implementing various authentication protocols in hardware and by making measurements of the system. Those could then be used to analyze and compare the resulting performance. This is very costly, as it is necessary to flash the ECUs each time another parameter set is to be tested and as the test of different configurations of architectures is very time consuming. That is why, it is essential to use simulations of those networks in order

to analyze the performance of those authentication protocols. Network simulators can emulate any architecture of internal automotive networks and the configuration of its components with minimal effort.

1.2 Problem and proposed Solution

Existing simulators are already able to emulate internal automotive networks and their components. Simulators, such as OMNet++ [MMT+13] or OPNET Modeler [Riv15], allow to specify the architecture of the network and the communication protocols that are applied on each layer of the OSI reference model. They allow to configure standard automotive components including ECUs, gateways and buses. While those components do implement the behavior of automotive systems, by default they do not provide mechanisms to implement the temporal behavior of security mechanisms based on realistic timing values. Thus, for instance the times to encrypt or decrypt messages within automotive networks are not considered in those approaches. Consequently, per default, the timing behavior of those networks cannot be analyzed depending on the parametrization of the chosen algorithms that are used within the authentication protocols.
That is why in this thesis a discrete-event simulator is proposed that makes it possible to easily vary the security configuration of an intra-vehicular network based on realistic time values.
The proposed simulator is highly modular and thus, enables to create and configure components in an automotive network, as well as their architecture. In contrast to other simulators, it is focused on a modular temporal and conceptual analysis of security protocols in internal automotive networks. Consequently, this simulator provides realistic timing values for cryptographic operations including hashing, random number generation, symmetric and asymmetric encryption and decryption, signing, verification and key generation. The timing values of those operations depend on the data sizes that are sent in the messages and the algorithms that are used for those operations. Thus, in order to achieve realistic results for all cryptographic operations, measurements were taken on a STM32 microcontroller, which is a representative hardware unit for the evaluation of state-of-the-art ECUs. Those measurements are then integrated within the simulator.
This makes it possible to validate various authentication protocols with different configurations, including in particular the cryptographic algorithms and the parameterization that are applied. In addition to that, by applying those mechanisms, different authentication protocols can be compared and also optimal configurations can be found in order to satisfy temporal constraints of the system. Additionally, the predefined security mechanisms and provided interfaces can be used to implement and validate user-defined protocols on the Application, Transport or Data Link Layer. Lastly, those layers are independent from each other. Thus, if different protocols need to be compared in the same scenario, it is sufficient to switch the layer implementation that comprises the corresponding protocol. This enables a maximum degree of comparability between various protocol implementations.

1.3 Outline

The work is structured as follows. In Chapter 2 the background for this thesis is given. It comprises automotive networks that are explained in Section 2.1, Controller Area Networks (CANs) and Controller Area Networks with Flexible Data-Rate described in Section 2.2, as well as security aspects presented in Section 2.3. Next, in Chapter 3 research underlining the necessity for security in automotive applications is given in Section 3.1 and existing simulators are discussed under Section 3.2. This is followed by the description of the measurements that were performed on the STM32 microcontroller in Chapter 4, which comprises the environment in Section 4.1, the realization in Section 4.2 and the data validation in Section 4.3. After that, the proposed simulator is presented. This includes the implementation in Chapter 5 containing the model core in Section 5.1, implementations based on this core in Section 5.2, possibilities to configure the simulator in Section 5.3, the Application Programming Interface (API) to access the model in Section 5.4, the I/O-processing in Section 5.5 and the Graphical User Interface (GUI) to visualize results in Section 5.6. Next, tests and optimizations are described in Chapter 6. This includes a test case generator that is presented in Section 6.1, tests that are performed to verify the software described in Section 6.2 and optimizations described in Section 6.3. In Chapter 7, then, the integration of the proposed simulator to an existing simulator [Mee15] for BMSs is described. In this chapter BMSs are described in Section 7.1, the existent simulator is presented in Section 7.2 and the integration with the proposed simulator is given in Section 7.3. In Chapter 8, under Section 8.1 the outcomes of performance measured for encryption are discussed, before the performance of the simulator is discussed in Section 8.2. Lastly, in Chapter 9 a conclusion is given and future work is presented.

In this chapter a short overview of security concerns in internal automotive networks is given. It is stated that the problems occurring in those networks can be tackled by applying authentication protocols. As those protocols are not yet established in cars, they have to be analyzed. This can be done by using the simulator that is introduced in this thesis. Next, in order to understand the components of this simulator, background knowledge about both automotive networks and security have to be provided. This is done in the following chapter.

2 Background

In order to grasp the mechanisms that form the IVNS, the following chapter introduces the relevant terms.
As this simulator aims to embody an automotive network, in Section 2.1 and Section 2.2 the composition and the concepts of automotive networks are described. A focus is put on Controller Area Network with Flexible Data-Rate (CAN FD) as this system is exemplarily realized in the proposed simulation. On top of that, several security mechanisms are implemented on the simulator. Thus, essentials of cryptography and secure networks are described in Section 2.3. Lastly, the simulator itself is based on a discrete-event simulation framework. An introduction to this topic is given in Section 2.4.

2.1 Automotive Networks

Automotive networks are categorized in external and internal communication systems [ZS11]. Both types are explained below. Next, in Section 2.1.1 components of internal automotive networks are introduced, before in Section 2.1.2 communication principles and in Section 2.1.3 common bus types of those networks are described.

External networks: This class stands for connections of vehicles to their environment over wired or wireless interfaces [ZS11]. It includes wireless communication of cars to infrastructure (V2I) or to each other (V2V) [BK12]. Those types of networks can be applied to avoid accidents or to exchange information about current traffic conditions.

Internal networks: Internal (a.k.a. intra-vehicular) networks refer to the inner communication network of a vehicle [ZS11]. This network type comprises the sensing of data in a car,

as well as the control of vehicle components via actuators. Also, those networks are used to exchange data between sensors and actuators in order to enable the control of the car.
As the implemented simulation model is based on the structure of internal networks, in the course of this work, the term automotive network and the following discussions refer to internal communication systems only.

2.1.1 Overview

Modern intra-vehicular communication networks consist of three types of components [MV13]. In conventional modern cars those are ECUs, buses and gateways. They are explained hereafter.

ECUs: ECUs are powered embedded devices that are linked to sensors and actuators [MV13]. While sensors gather data about the vehicle state, actuators are used to perform actions. As most functionalities in cars are enabled by data exchange between multiple components of this type, their corresponding ECUs are connected via buses [ZS11].

Bus systems: Buses realize the connection between ECUs and are categorized according to their software and hardware properties, which are described in Section 2.1.2. Depending on the field of application, diverse bus types are applied in modern cars [ZS11]. Those are the frequently used CAN, FlexRay, Local Interconnect Network (LIN) and Media Oriented Systems Transport (MOST) networks as well as the upcoming Ethernet and CAN FD communication systems [TGH+15]. They are introduced in Section 2.1.3 and Section 2.2.

Gateways: Gateways are used to interconnect communication systems [ZS11]. As various bus technologies are incompatible with each other, gateways translate signals between bus systems. The gateway follows the defined sending and receiving mechanisms for each bus.

Whereas in the 1990s automotive networks consisted of about 8 to 10 ECUs, in 2009 in a modern car this number increased tenfold [BK12]. This trend is very likely to last. Continuously upcoming innovations, such as Advanced Driver Assistance Systems (ADAS) or lane departure control will require more components in the future.
To handle this complexity, communication systems in modern cars are divided into subsystems [ZS11]. Each of those subsystems is optimized for a certain type of application and does therefore apply a bus technology that is suitable for its purpose. Those bus systems exchange data via gateways [ZS11]. This is shown in Figure 2.1. Along with different applications for those subsystems, distinct requirements are needed for them [LHD99]. Accordingly, based on those requirements, bus technologies are categorized into four groups, that were defined by the Society of Automotive Engineers (SAE) [LHD99].
Those are *Low-Speed Systems (Class A)*, with data rates of up to 10 kBits/s. They apply LIN buses and are cost-efficient but unreliable. Thus, non-safety critical signals are exchanged by buses of this type.
In *Driving Assistance Systems (Class B)* for the transmission of general information data rates

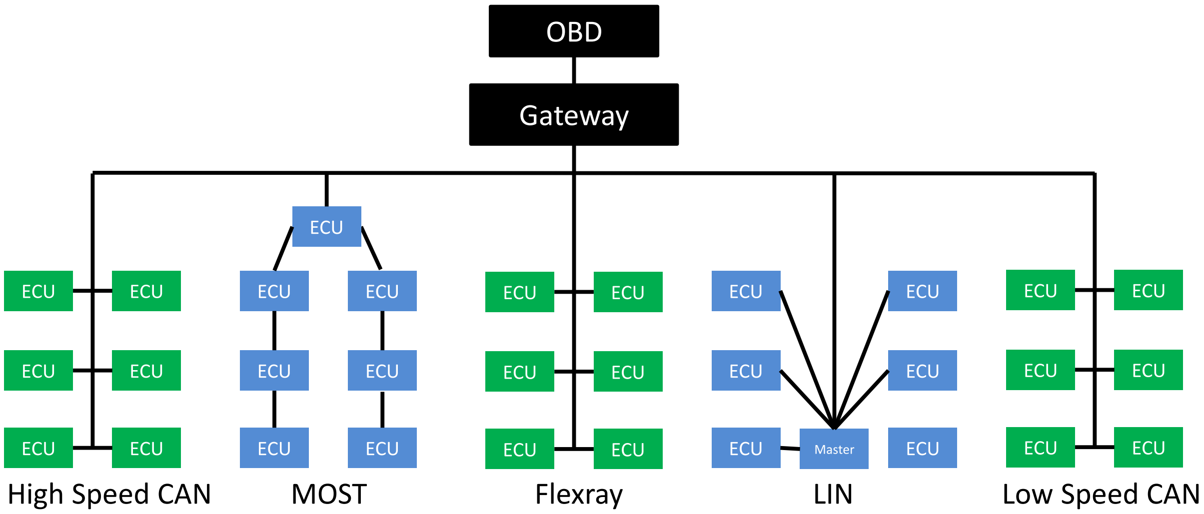

Figure 2.1: Simplified automotive network architecture. Based on [TGH+15].

between 10 kBit/s and 125 kBit/s are used. In those systems both reliability and latency are important, but not safety critical factors. Thus, CAN and LIN buses are used.
In safety critical control tasks that require real-time data transmission, *High-Speed Systems (Class C)* are utilized. Those subsystems communicate messages with high frequencies, short latencies and high reliability. For instance, they are used to control the engine, the transmission, the brakes or the chassis of the car. In those structures high-speed CAN and FlexRay communication systems are employed which achieve data rates between 125 kBit/s and 1 MBit/s.
Infotainment Systems (Class D) with functionalities needed in entertainment devices, such as the transmission of audio-video data, require data rates of up to 1 GBit/s. In comparison to Class C systems, however, reliability and latency are not critical parameters. Nowadays, Ethernet, MOST and Low Voltage Differential Signaling (LVDS) are used in those systems.

2.1.2 Communication

Bus technologies differ in terms of their communication characteristics and their placement in the OSI reference model. The most important of those characteristics are listed below.

Characteristic properties

Topology: Three types of topologies are utilized in automotive communication systems [ZS11]. The one that is prevalent in cars is the bus topology. It is defined as a wire with multiple ECUs connected to it as it is depicted in Figure 2.2. In a star topology, several devices are connected to one central star coupler or ECU. This instrument is responsible to manage the communication between the linked units. In a ring topology all participants of the communication are joint together to form a circle. Each ECU is connected to two neighbors and messages are sent by circulating them along the ring. Illustrations of those topologies are shown in Figure 2.2.

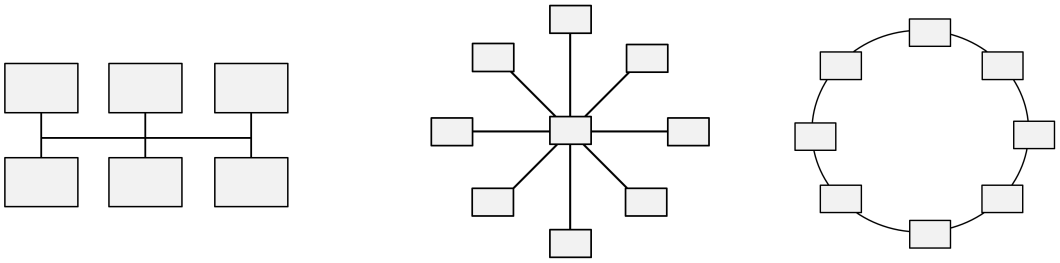

Figure 2.2: Different types of topologies. Adapted from [ZS11]. *left to right:* Bus Topology, Star Topology, Ring Topology

Communication model: The communication model comprises the connection type and the used addressing [ZS11]. The connection type is the distinction between connection-oriented and connectionless communication [ZS11]. The former is performed as a three-part process. In the initialization phase, a connection between two partners is established. Next, the ECUs start to exchange data and end the communication then. Contrary to that, messages in a connectionless protocol are sent spontaneously without any agreed upon connection. Solely the addresses decide about whether a message is processed by a communication unit [ZS11]. This addressing is done either based on the ID of the target device or the content type of the message.

Data transmission: When data is transmitted, this is either done in a synchronous or an asynchronous way [ZS11]. In synchronous transmission, a common clock is used for synchronization, while asynchronous transmission does not have a common schedule.
On top of that, depending on the protocol that is implemented, messages are sent to certain units (multicast) or to all units in the network (broadcast) [ZS11].
In a client-server communication [ZS11] clients request messages that are answered by a server. Contrary, in a publisher-subscriber environment [ZS11] components, that wish to receive messages from a sender, subscribe to it. Then, messages of this sender are directed to its subscribers. Depending on the used communication scheme, information is sent either after fixed time intervals (time-triggered) or once events occur in a node (event-triggered).

Bus access method: As there are multiple ECUs connected to a bus, collisions can occur if several units start to send in close intervals. Those conflicts are resolved variously [ZS11]. Event-triggered networks include approaches such as polling, tokens and Carrier Sense Multiple Access (CSMA), whereas time-triggered systems rely on Time Division Multiple Access (TDMA) [ZS11]. In the following the most important mechanisms are presented.

- *CSMA Collision Detection (CSMA/CD)* [TW11] requires the communicating units to listen to the bus. Subsequently, sending is started if the carrier is sensed free. While sending, transmitting units also listen to the bus. Once a collision occurs, a jammed signal is detected at the sender and the transmission is restarted after a random waiting time.

- *CSMA Collision Avoidance (CSMA/CA)* [TW11] systems have units that listen to the bus. Once they sense the bus free, they wait for a random time. Optionally, the channel is re-

quested. If the request is successful, the receiver acknowledges the connection establishment to the sender. At the same time it notifies all other participants about the reservation of the channel and the expected duration of the communication. If no acknowledgment was received at the sender's side after a defined time, the sender will back off for a random time and retry to acquire the channel then. The random interval for the back-off time is increasing with the number of failed connection attempts.

- *CSMA Collision Resolution (CSMA/CR)* [CMLL13] systems have units that sense the bus, wait an idle time and start their sending process once the bus is free. If two participants start to send at the same time, the content of the message decides which potential sender's packet is transmitted. This mechanism is described in detail as Arbitration in Section 2.2.

- *Time Division Multiple Access (TDMA)* [ZS11] slices the communication channel into uniform time slots. In the synchronous approach [ZS11], each time slot is assigned to a sender. Due to this fixed assignment, no collisions can occur and every transmitting unit is able to send at a constant bit rate. Asynchronous TDMA [ZS11] extends this approach. It fills unused time slots with further messages that are put on the bus using multiplexers. In order for the receiver to be able to recover the affiliation between stream and packet information, a further header is added by the multiplexer. This enables the demultiplexer to reassign the asynchronously transmitted message streams at the receiver's side.

ISO/OSI Reference Model

In order to classify the tasks of communication protocols, the International Organization for Standardization (ISO) introduced the Open Systems Interconnection (OSI) reference Model [DZ83]. It consists of a structure of seven layers [DZ83], where each layer covers certain tasks in the communication (see Figure 2.3). Lower layers of the model offer services to higher layers. This makes it possible to abstract the layers from their implementation. Hence, it enables to interchange various layer implementations.
In automotive networks all seven layers are present in MOST systems [Con11]. Due to less complex communication processes in the remaining automotive bus systems, for them this model is reduced to four layers [Con11]. Those are the following.

Application Layer: This layer [DZ83] is used to offer services to applications and to handle transmissions by exchanging data with the next lower layer. For transmission this level exchanges packets with the Transport Layer.

Transport Layer: This layer [ZS11] ensures that incoming data is prepared to be sent on lower layers. Thus, it is used to chunk incoming packets into segments and to put them in the defined format for the Data Link Layer [ZS11]. According to [ZS11] this is done as defined in the standards FlexRay TP, SAE J1939, TP 1.6 and TP 2.0, DoIP and ISO TP, which is presented in Section 2.2.2. The resulting segments are passed to the Data Link Layer.

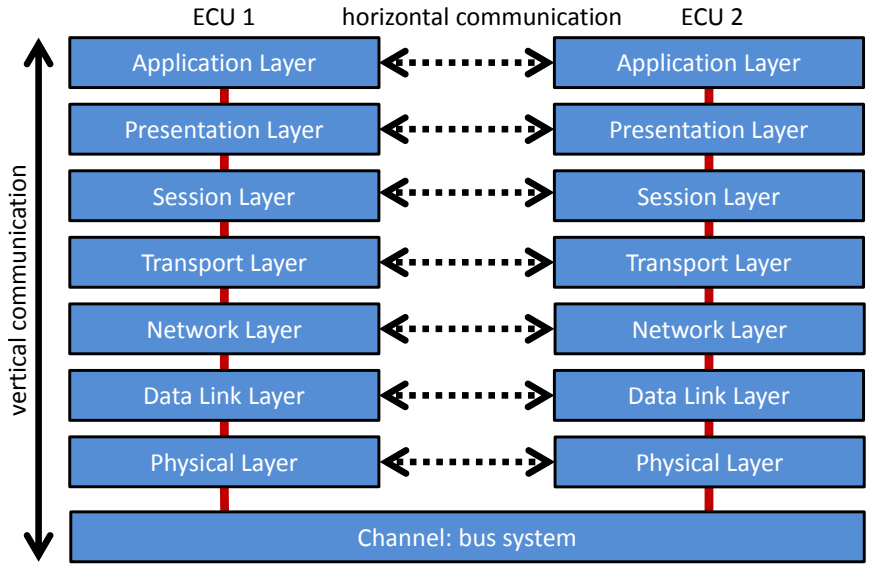

Figure 2.3: OSI reference model. Adapted from [DZ83].

Data Link Layer: This layer has three tasks [ZS11]. First, the bus access control described earlier. Next, this is fault detection in transmitted frames. On top of that, addressing is performed on this layer. This is done either based on content or unit addresses.

Physical Layer: According to [ZS11] the Physical Layer is defined by the signaling characteristics and the corresponding physical tools, that are used for transmission. The tools, as well as the sending process vary dependent on the bus technology that is used .

2.1.3 Buses

Bus technologies are characterized based on the descriptions in Section 2.1.2. They include the CAN, LIN, FlexRay, MOST, CAN FD, and Ethernet systems. Except for CAN and CAN FD, which are treated in Section 2.2, the functioning of those bus types is introduced hereafter.

Local Interconnect Network: The first version of LIN [LIN10] was released by the LIN Consortium in 1999 and extended until 2010. Its main intention is to keep the costs per bus low, compared to the remaining bus types.
A LIN bus consists of one master node and several slave nodes. The former contains master and slave tasks, while the latter accommodates slave tasks. Master tasks decide about the timing and frame type of the transmitted message and slave tasks about their content. Based on TDMA and selectable scheduling tables, the master task prompts the slave tasks regularly by sending header messages to them. Depending on the header's frame ID, they respond with a data frame of up to 8 bytes, which is read by all slaves on the bus. Four types of frames are defined. Those are unconditional frames that contain signals transmitted in a time-triggered way, event-triggered frames based on events, sporadic frames that share time slots and diagnostic frames used for diagnosis. Data rates are of up to 20 kBit/s.

FlexRay: Based on the rising number of real-time applications at the end of the 1990s, efficient and rapid bus systems had to be developed [ZS11]. That is the reason why at the beginning of the 21^{st} century, the FlexRay Consortium was found and the FlexRay protocol was specified and standardized in ISO 10681-1 and ISO 10681-2. This protocol is defined as follows.
As FlexRay [Fle05] uses TDMA for communication, ECUs' clocks in the network are globally synchronized using a time basis. Also, the sending process is running in cycles that are divided into static and dynamic segments. Static segments are divided into slots that are reserved to particular ECUs. Those use them to send messages. Next to those kinds of frames, dynamic segments can be used to send additional messages on an irregular basis. In applying those concepts with frame sizes of 255 bytes, data rates of 2.5, 5 and 10 MBit/s are achievable.

Media Oriented Systems Transport: As part of the addition of infotainment systems to automobiles in the 1990s, new components have come into play, that require flexibility, interchangeability and high data rates [MOS05]. For this reason MOST was introduced.
According to [MOS05], MOST systems are TDMA-based and consist of up to 64 slaves of which 4 are declared as masters. The timing master is responsible for the synchronization of the individual network devices. It is complemented by the network master that sets up the network, the connection master that initiates the synchronous communication between devices and the power master that is responsible for the power supply. Additionally, MOST devices consist of function blocks that are able to communicate either with their application or with function blocks within the MOST network. This is done over three communication channels. Those are the control channel that administers connections, the synchronous channel that transmits high-speed data and the asynchronous channel that handles burst-like overhead by offering additional bandwidth. Data rates of 25, 50 and 150 MBit/s are achieved with payloads between 372 to 1506 bytes per message.

Automotive Ethernet: Due to incompatibility and high costs in current automotive standards, Ethernet was proposed for automotive networks. As it is state-of-the-art in computer networks, it is produced in high numbers and is thus, cheap in production and development [ZS11]. Nevertheless, IEEE 802.3 Ethernet, as it is applied in current computer networks, is rarely deployed in automotive networks as it is missing real-time capability [IXI14].
That is why various proposals were introduced, although none was standardized yet. However, recently the "BroadR-Reach" technology [IXI14] for the Physical Layer of Ethernet is most common and highly likely to be standardized. In contrast to other Ethernet types, it uses a Single-Twisted-Pair cable that reduces the harness weight and the cost per connection. It offers full duplex connections to send and receive data with data rates of up to 100 Mbit/s. On the Data Link Layer mostly the standards grouped as Audio Video Bridging (AVB), including IEEE 802.1Q and IEEE 802.1 AS are common [IXI14]. Furthermore, the Time-Sensitive Networking (TSN) [IXI14] standards that are introduced by the TSN Task Group are applied in those networks. Those include the successors of the AVB standards. Further proposed standards [ZS11], that can be found in current automotive networks, are ProfiNet and TTEthernet. On higher layers the Some/IP protocol is common.

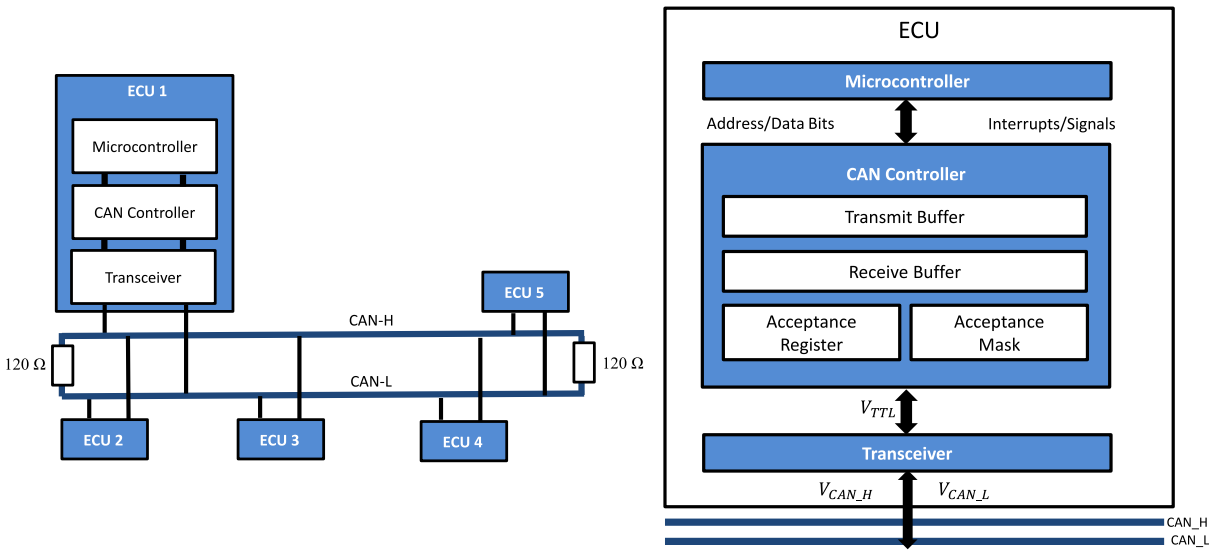

Figure 2.4: *left:* Example architecture: CAN Bus with 5 ECUs on Physical Layer. *right:* Detailed view on the components of an ECU. Adapted from [ZS11].

2.2 Controller Area Network

CAN was first introduced in 1986 by the Robert Bosch GmbH with the intention to enable reliable real-time communication between ECUs in an automotive network, while keeping the cost per connection low [Vos05]. According to [Vos05], it was standardized in 1995 in the ISO 11898-1 and ISO 11898-2 standards. It enjoys a high degree of popularity as it is cheap, fast and reliable at the same time [Vos05]. Thus, hereafter, the structure of CAN is presented and CAN FD is introduced.

2.2.1 Physical Structure

The Physical Layer of CAN is composed of the bus topology and hardware components, as well as of the physical communication process [Vos05]. The bus topology is formed by ECUs, buses and bus couplers. These components interchange data on a physical level by applying a defined data transmission mechanism. Both components and their mechanisms are introduced below.

Bus components: CAN buses consist of a conventional bus topology as it is described in Section 2.1.2. This bus links multiple ECUs by connecting them to two wires which are called CAN-H and CAN-L (see Figure 2.4 left) [ZS11]. Those two cables are terminated by a resistor that is between 100 and 130 Ω.
ECUs in those systems consist of a microcontroller, a CAN controller and a transceiver [ZS11]. This is illustrated in Figure 2.4 on the right side. While the microcontroller is responsible for the time and content of the transmitted data, the other two components implement the methods to exchange data between ECUs. Therefore, the CAN controller is connected to the microcontroller and the CAN controller is linked to the transceiver that is attached to the bus.

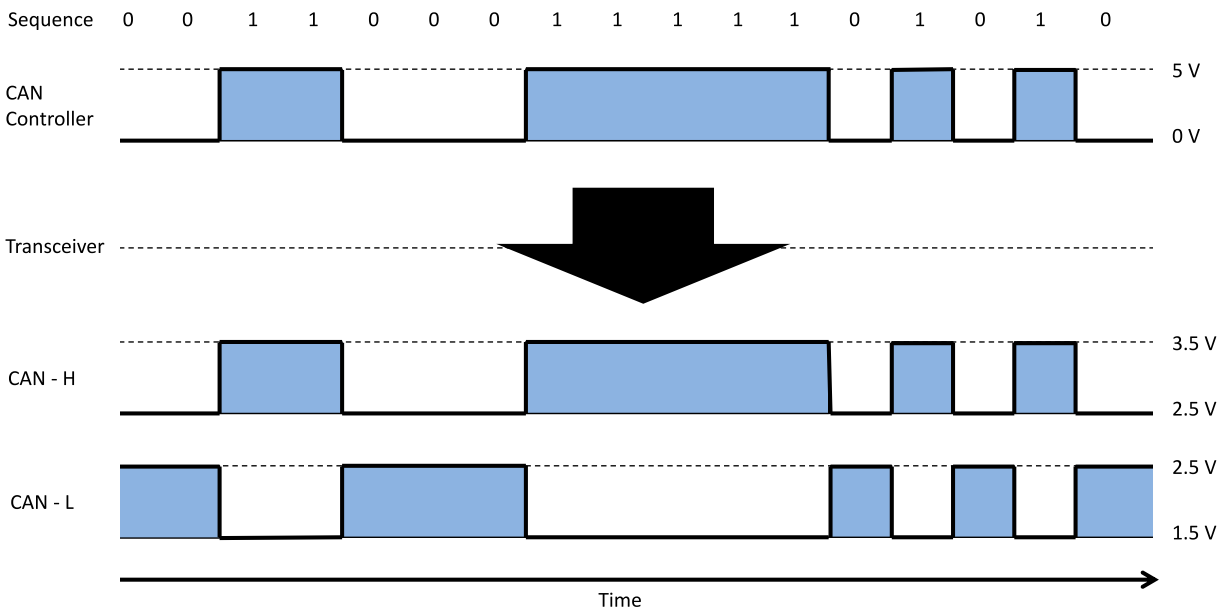

Figure 2.5: ECU sending bit sequence on CAN bus resulting in voltage levels on CAN-H and CAN-L. The CAN controller's TTL-level is coded using NRZ-coding. Adapted from [Vos05].

Transmission process: In an ECU data transmission is initialized by the microcontroller [ZS11]. When its application is willing to send [ZS11], it pushes the message to the CAN controller. Subsequently, the CAN controller chooses the highest priority message and initiates its transmission. Analogously, received messages are pushed up to the application.
The physical transmission works as follows [Vos05]. Messages consist of a series of dominant (= 0) and recessive (= 1) bits. The bus differentiates those two states by setting the CAN-H voltage to 3.5 V and the CAN-L voltage to 1.5 V in a dominant state and by setting both wires to 2.5 V in the recessive state.
To put a bit sequence on the bus the transceiver unit varies those states in time. If all ECUs on a bus set their level to a recessive state the bus is recessive. In all other cases it is dominant (see Table 2.1) [Vos05]. For transmission a Non-Return-to-Zero (NRZ) coding is used (see Figure 2.5) [Rob91] and bit-stuffing is applied in order to generate edges for synchronization [Rob91]. Bit-stuffing on the sending side inputs a bit of inversed polarity when five bits of same polarity occurred in a row. The receiver reverts this by removing the bits again.

2.2.2 Higher Layer Specification

The specification of the CAN bus includes the Physical Layer, that was introduced in Section 2.2.1, and the specifications of the Data Link Layer, the Transport Layer and the Application Layer, which are explained in the following.

Communication principle: In CAN, up to 8 bytes are transmitted with a data rate of up to 1 MBit/s [Rob91]. Hereby, every ECU is allowed to send at any point in time [Vos05]. Nevertheless, usually messages are sent periodically with a fixed time interval as part of a

Time Slot	ECU 1	ECU 2	ECU 3	ECU 4	Bus State
0	0	0	0	0	0
1	1	1	1	1	1
2	0	1	0	0	0
3	0	0	1	1	0
4	1	1	1	0	0

Table 2.1: Multiple ECUs on one bus setting their output level to dominant (= 0) or recessive (= 1) in sequential time intervals and resulting state on the bus. Adapted from [Vos05].

message stream. Each stream is defined by a message ID and has one sender, as well as one or multiple receivers [ZS11]. This identifier is sent within the header of each frame. In data and remote frames the message ID decides if a receiver ignores or processes this message.
Throughout the communication, active units are listening to the bus [Vos05]. Once they detect a transmission, they switch to receiving mode and wait until it is finished. If ECUs want to send a message, they sense the bus and send the message after the bus is idle for an Inter Frame Space (IFS) of three bit times. In case of simultaneous bus access by multiple sending devices messages with lower identifiers are sent as they have a higher priority.

Message frame: Four kinds of frames are transmitted in CAN [Vos05]. For the most part of the communication data frames are sent, as they are used for transmission of bus signals. Next to those message types, remote frames are utilized to request data from other parties. Furthermore, error frames handle faulty data on the bus and overload frames signal when a node can not handle any more incoming data.
Data and remote frames are shaped identically [ZS11], except for the data field that is not present in the latter. Their scheme is shown in Figure 2.6.

- *Start of Frame (SOF) and End of Frame (EOF)*: The dominant SOF bit marks the beginning of the transmission procedure, while the 7 EOF bits mark its end [Vos05].
- *Arbitration field*: The Arbitration field consists of the Message ID and the Remote Transmission Request (RTR) field [Vos05]. Arbitration is performed based on those parameters. The message ID indicates the type of data that is sent, while the RTR field indicates the sort of frame, which is either data or remote type. In CAN 2.0A the message ID consists of 11 bits and in CAN 2.0B of 29 bits (extended frame).
- *Control and Data fields*: The control field indicates the length of the transmitted data and pinpoints if the extended message ID is used [Vos05]. The data field contains the payload.
- *Cyclic Redundancy Check (CRC) and Acknowledgement (ACK) fields*: To verify the data, upon reception of a frame, a CRC check is performed on sending at the sender side and during reception on the receiver side [Vos05]. Upon reception of the frame, the result of this check is compared and the ACK field is set to recessive by the receiver, if the check was successful. Else a dominant bit is output.

Figure 2.6: CAN Data Frame as defined in ISO 11898-1. Adapted from [Vos05].

- *Inter Frame Space (IFS)*: This is the minimum time that needs to pass after one frame finished and another one can be sent [Vos05]. It comprises 3 bit times.

Arbitration: If multiple ECUs try to access the bus simultaneously, arbitration is performed in order to decide which unit is allowed to send its message [Vos05].
With regards to the physical behavior, a dominant bit overrides a recessive bit when sent at the same time [Vos05]. When a bit is put on the bus, each sending ECU checks if the value of the bit conforms to the current state of the bus [ZS11]. This process is known as bus monitoring. If multiple participants send at the same time, their fields are compared bit per bit in the order they are sent. Hence, during the outputting of the arbitration field bits by the senders, the frame with the lowest message ID (= highest priority) overrides all other messages [Vos05]. Due to bus monitoring, the senders with the lower prioritized frames realize that their transmission is not successful. Consequently, they stop sending and switch to listening mode. Once the bus is idle again, retransmission of the same frame according to this scheme is executed anew [Vos05].
This procedure does not lead to conflicts because each message ID is only transmittable by one sender. Similarly data and remote frames do not collide as the RTR field is part of the arbitration field and is 0 in the former and 1 in the latter case [Rob91].

Transport Layer Segmentation: On the Transport Layer of the OSI reference model, the protocols TP 1.6, TP 2.0, SAE J1939 and the ISO TP are employed in CAN [ZS11]. Yet, only ISO TP [Int11] was standardized under ISO 15765-2. That is why this protocol is described in the following.
CAN frames are only capable of transmitting 8 bytes per frame. If a data amount that is greater than that is sent, the sender segments the data into normal CAN frames using the ISO 15765-2 standard [Int11]. The receivers then gather all incoming frames and reassemble them to recover the original data. All data that is to be sent is transmitted in either of four types of frames [ZS11], that are illustrated in Table 2.2.
According to [ZS11], this works as follows. In all the CAN frame is left unaltered. Solely the data field of the frame is changed. The first 2 bytes of this field are redeclared to form the Protocol Control Information (PCI) field.
If the data that is to be sent does not exceed 7 bytes, the data is transmitted unsegmented in a *Single Frame*. The PCI bits are set as shown in Table 2.2. In case an application is willing to transmit data between 8 and 4095 bytes, data segmentation needs to be applied on the Transport Layer. Therefore the data is chunked into multiple smaller segments. The first of those chunked

Bit Offset	0...3	4...7	8...15	16...23	24 ... frame end
Single	0	Size (0...7)	Data A	Data B	Data C
First	1	Size (8...4095)		Data A	Data B
Consecutive	2	Index (0...15)	Data A	Data B	Data C
Flow	3	FC flag (0,1,2)	Block Size	ST	

Table 2.2: Frame types and their structure according to the ISO 15765-2 standard. Adapted from [Bec15].

frames is called *First Frame* and contains the initial segmentation information and is marked to indicate the incoming data length, as shown in Table 2.2. All further pieces of the message are sent in *Consecutive frames*. This type of frame is marked with the tag 2 and uses four bits to indicate the sequence number of the frame as is shown in Table 2.2. They transmit the actual data chunk with 7 bytes of size. As this procedure increases the traffic on a CAN bus, a mechanism is defined to control the number of sent messages on the bus. This is done by using *Flow Control Frames* that are exchanged throughout the communication.

2.2.3 CAN with Flexible Data-Rate

Two major drawbacks of CAN are its bandwidth limitation to 1 MBit/s and its maximum payload size of 8 bytes per frame [Har12]. Those disadvantages are addressed in the CAN FD protocol [Har12]. It increases the bit rate on the one hand and allows payloads of up to 64 bytes on the other hand. At the same time the integration to existing CAN technologies is cost efficient, as only the CAN controllers need to be enhanced to support CAN FD.

Data transmission: When a message is processed in a CAN controller, it is possible to switch to a faster bit time within one CAN frame during sending [Har12]. This is done in CAN FD. In order to be able to send more than 8 bytes, during the data phase of the frame a predefined higher sampling time is applied [Har12]. Compatibility to ordinary CAN buses is kept by leaving the sampling time at the defined CAN speed outside of the data phase. This stage is called the arbitration phase [Har12].

CAN FD frame: CAN FD frames are dividable into three parts as shown in Figure 2.7 [Har12]. The first arbitration phase consists of the same components as a CAN frame and uses the standard CAN speed. Nevertheless, four fields of this phase are redefined. The Extended Data Length (EDL) field indicates whether a CAN or a CAN FD frame is sent. Furthermore, the reserved bits r0 and r1 are transfered dominantly for synchronization. The Bit Rate Switch (BRS) field is dominant, if the bit rate of the arbitration phase is similar to the bit rate of the data phase. Furthermore, to handle errors, the Error State Indicator (ESI) field is added. Likewise the data length field is used to define the transmitted payload size of up to 64 bytes [Har12]. Due to CAN FD's timing behavior those values are restricted to 12, 16, 20, 24, 32, 48 and 64 bytes [Har12].

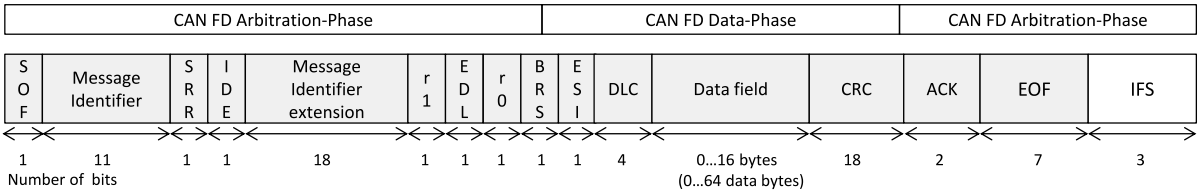

Figure 2.7: CAN FD Data Frame as defined in ISO 11898-7. Adapted from [Har12].

So far all basics of automotive networks are presented. This is done by first discussing the characteristics of automotive networks. Also, the components of the network, the OSI reference model and the various bus types are introduced. Lastly, CAN and CAN FD are discussed in detail. Thus, after all essentials about automotive networks are discussed, next, the second component of this thesis' topic is introduced. This is the security aspect of communication in networks.

2.3 Secure Networks

Networks consist of several systems and subsystems that are connected over a medium. In case of automotive networks, those connections are insecure and provide the system with a target to hackers. This danger is reinforced, as the system is connected to networks, which provide interfaces to the outside world. By exploiting those interfaces, attackers can intercept the communication. If that happens, security critical data can be modified and read along or safety endangering data can be injected. Therefore security mechanisms need to be implemented. In the following, first, basic terms of security are discussed. Next, an introduction to cryptography is given. Based on that, then, Authentication and Authentication Protocols are presented.

2.3.1 Basic Terms

First, characteristics that form secure systems are introduced. Those are protection, privacy, dependability and reliability [Eck08]. Their aims are confidentiality, integrity and availability [Eck08].

Security components: A network is said to be secure, if it provides the following four components [Eck08]. The first of them is *protection*. It includes the shielding against unauthorized manipulation of system resources and methods to prevent loss of data. The term *privacy* is the capability of a person to control the flow of information that is of concern to it. Especially no information must be given away to other parties unwillingly. *Dependability* is the guarantee that a system will not change to an illegal state and thus, is fully functional throughout its lifetime. Additionally, *reliability* implies that all specified functions of a system are executed solidly and as expected.

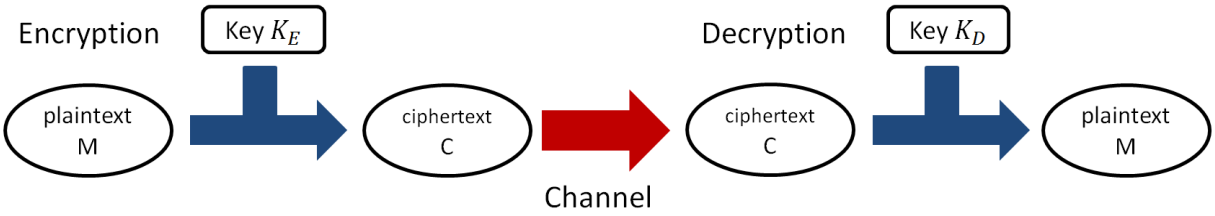

Figure 2.8: Example of a cryptosystem. A sender encrypts a message with key K_E and transmits it over a channel to the receiver. The receiver decrypts the message with key K_D and can read the message. Adapted from [Eck08].

Protection targets: The three protection targets of security are confidentiality, integrity and availability [Bis05]. *Confidentiality* is the secrecy of information or resources. By not disclosing intelligence, safety imperiling knowledge transfer to unauthorized people is avoided using access control methods. A further aim of security is *integrity*, which is the trustworthiness of data. As unauthorized persons could modify a message during transmission, data and origin integrity need to be ensured. The former means that receivers need to make sure that the content was unchanged by a third party during sending. Additionally, the latter implies that it has to be ensured that the message was sent by the expected sender. In the same way *availability* of a system needs to be guaranteed to enhance the reliability of it. Malicious hackers could attack the system and make it or parts of it unavailable. Consequently, this could cause major incidents in safety critical applications.

2.3.2 Cryptography

To be able to control the access of data and to hide content from others, cryptography is used [TW11]. Cryptography is the art of concealing meaning and is used in security networks as a supporting tool [TW11]. This is done by encrypting plain data bytes (plaintext) using defined encryption functions, that are parametrized by a key [Eck08]. The output is a modified version of the original data (ciphertext). This ciphertext can be decrypted using another key, that is either equal or different to the one used for encryption. Those kinds of systems are called cryptosystems [Eck08] and are depicted schematically in Figure 2.8. According to [Eck08], mathematically those systems KS are described by the tuple

$$KS = (\boldsymbol{M}, \boldsymbol{C}, EK, DK, E, D), \tag{2.1}$$

where $\boldsymbol{M}$ represents all plaintexts, $\boldsymbol{C}$ all ciphertexts, EK all keys used for encryption, DK all keys used for decryption, E the encryption method and D the decryption method.

The bijection between all encryption keys EK and all decryption keys DK is defined as

$$f : EK \rightarrow DK \tag{2.2}$$

which links an encryption key $K_E \in EK$ to a decryption key $K_D \in DK$ using $f(K_E) = K_D$.

The encryption and decryption methods create the cipher and clear texts using

$$E : \boldsymbol{M} \times EK \rightarrow \boldsymbol{C} \tag{2.3}$$

$$D : \boldsymbol{C} \times DK \rightarrow \boldsymbol{M} \tag{2.4}$$

with $\forall M \in \boldsymbol{M} : D(E(M, K_E), K_D) = M$.
If those types of systems use the same key for encipherment and decipherment, the system is called a classical or symmetric cryptosystem. In contrast to that, if different keys are applied for those methods, the system is called public-key or asymmetric cryptosystem [Bis05]. For the encryption mechanism, hereby, the secrecy of the keys is the only thing that is concealed. All other parameters used in the cryptosystem are known. This principle is called the Kerckhoff principle [Bis05].

Symmetric-key cryptography

Symmetric cryptosystems apply the principle described above. In this case all senders and all receivers agree upon a common secret - namely the key. In terms of above stated equations thus, $K_D = K_E$ holds. This key is used to encrypt the message at the sender side and used to decrypt it at the receiver sides [Eck08]. The security of this communication depends on the strength of the encryption mechanism and on the security of the common key [BNS05].
Symmetric ciphers are classified into substitution and transposition ciphers [TW11]. The former replaces each character of the plaintext with another character or word, while the latter rearranges the characters of the plaintext. On top of that, methods that encrypt one byte at a time are called stream ciphers. Conversely, mechanisms that chunk a plaintext into blocks and encrypt each of them individually to form a cipher, are called block ciphers [Eck08].

Advanced Encryption Standard: The Advanced Encryption Standard (AES) [NIS01] algorithm was invented in 2000 as a successor to the Data Encryption Standard (DES) algorithm. It is a symmetric block cipher, that uses key lengths of 128, 192 and 256 bits and block sizes of 128 bits for encryption [Eck08]. It is using both transposition and substitution mechanisms in order to generate a cipher in multiple rounds. Additionally, it offers a high degree of security as there are 2^{126} steps needed to break a 128-bit key and 2^{254} steps to break a 256-bit key [Eck08].

Public-key cryptography

Other than in symmetric cryptography, asymmetric or public-key cryptography uses different keys for encryption and decryption, following the same principle as explained above [BNS05]. Here a sender uses the public key K_{Pub}, that is accessible to any participant of the network to encrypt the message. This key has a corresponding private key K_{Priv}, that is known to the communication target and is used to decrypt the message. Similar to the symmetric case, the security again depends on the strength of the encryption mechanism and on the security of the common key.
Two important asymmetric encryption procedures are the Rivest-Shamir-Adleman (RSA) algorithm and Elliptic Curve Cryptography (ECC) [BNS05].

Rivest-Shamir-Adleman Algorithm: In the RSA [RSA78] algorithm the fact is exploited that there is no efficient way to solve prime decompositions for all numbers. Thus, prime multiplication is an one-way function. It is used to perform encryption and certification by mostly applying keys of 512, 1024 or 2056 bits. If applied correctly, it offers a moderate degree of security, as it is possible to determine the key with a chosen-ciphertext attack [BNS05]. Nevertheless, those attacks also need to perform multiple computation steps and with increasing key size, the key breaking is increasingly costly.

Elliptic Curve Cryptography: ECC [KKM11] is another asymmetric encryption method that was invented in 1980 and is based on elliptic curves. It uses mostly keys with sizes 192, 256, 384 and 521 bits. At the same time, in comparison to RSA, ECC requires smaller key lengths to guarantee the same security level. Thus, for instance to break a ECC key of 160 bit length similar resources are required as in the case of a RSA 1024 bit key [KKM11]. ECC's operations are more costly than the ones of RSA. Nevertheless, if the right parameter sizes are used it is more efficient than RSA as well.

Hashing mechanisms

A hashing function is a compression function, that maps a bit string of arbitrary length onto a bit string with fixed length [BNS05]. At the same time it is computable efficiently and is not invertible. The probability that an hashing mechanism outputs the same bit string for different inputs is called collision resistance [BNS05]. For security applications mechanisms with good collision resistance are preferred. Important hashing algorithms are the Message-Digest-Algorithm 5 (MD5), the Secure-Hash-Algorithm 1 (SHA1) and the Secure-Hash-Algorithm 256 (SHA256) [BNS05].
The execution of the MD5 algorithm is done in 4 rounds of 16 computation steps and produces an output of 128 bits [Riv92]. SHA1 produces an output of 160 bits where in 4 rounds 20 computation steps are undertaken [EJ01]. An extension to SHA1 is the SHA256, which only differs in the number of rounds and its digest length which is 256 bits [EJ01].

2.3.3 Authentication

Cryptography as it is defined in the previous section can now be used for authentication, which is described in the following.
Authenticity is defined as the genuineness of an object. It is verifiable using unique identifiers or a characteristic property [Eck08]. Authentication can be done using knowledge, possession or biometric characteristics. Tools for this process are Message Authentication Codes (MACs), signatures and certificates. They are applied in so called authentication protocols. Three of those protocols are introduced in Section 2.3.4.

Message Authentication Codes: MACs are used to verify the integrity and the authenticity of data or objects [BNS05]. In order to do so, both communication partners agree upon a common key. This key is used to encrypt the outgoing message whose integrity and authenticity are to

be proven. As a result of this encryption, a MAC is generated, which has the same properties as a hash function that was described above. This MAC is sent to the receiver together with the message. After the receiver gets the message, he can prove the authenticity of the message by calculating the MAC with the common key and by comparing it to the received MAC.

Digital signatures: Digital signatures are used to sign electronic documents in an unforgeable way [TW11]. In current communication systems symmetric-key signatures and public-key signatures are applied.
The former is based on symmetric communication and a Trusted Authority (TA) that handles the authentication (see [TW11]). In the latter case the signing is done asymmetrically at the sender side [TW11]. Here, it is assumed that both the encryption and decryption mechanism can be used interchangeably to encrypt and decrypt the message. So a sender A, that is willing to send a message, signs a plaintext by encrypting it with the private key K_{Apriv} resulting in $E(M, K_{Apriv})$. This cipher is sent to the receiver together with the clear plaintext. Next, the received digital signature is verified by using A's public key giving the plaintext M with $M = D(E(M, K_{Apriv}), K_{Apub})$. If the content of this message is verified against the one resulting from the sent plaintext, the message can be trusted.
In order to keep the signatures small, this plaintext M is hashed, before it is signed and sent [TW11]. Thus, any receiver that wants to verify a message first hashes the received plaintext M and compares it to the hash that is resulting from the signature.

Certification: According to [Eck08], a digital certificate is an attestation of the association of a public signing key to a real person. They are used to receive a trusted public key from a communication partner. A certificate is only issued by trusted Certificate Authorities (CAs), which enable any receiver of a certificate to verify its authenticity securely [Bis05].
The main content of a standard certificate of X.509 format includes the following items [Eck08]. Those are key information including the user's public key and the algorithm used for encryption. Also, the user name, the Certificate Authority (CA) issuing this certificate and a digital signature are part of it. The digital signature is generated by the CA and is authenticated in the way described above. This verification process can be done in two ways as follows [Eck08].
If the sender and the receiver are under the same CA, both have access to the public key of it. Thus, the receiving party can verify the sender certificate by authenticating its signature.
In contrast to that, it is also possible that two certificates were issued by different CAs. In this case cross-validation is applied.

2.3.4 Authentication Protocols

In the previous section, tools for authentication are described. Those tools are used in authentication protocols which are discussed in this section.
For current automotive real-time applications buses are used. To ensure safety on those buses certain requirements have to be taken into account. First, devices in those systems have less computational power. Thus, symmetric encryption mechanisms are preferred in those proto-

cols, as asymmetric methods require excessive processing times. On top of that, frame sizes are restricted and data rates are comparably slow in those networks. Thus, a small overhead is demanded from secure protocols as well. The three protocols LASAN, TESLA and TLS, that are proposed to tackle those requirements, are presented in the following.

Lightweight Authentication for Secure Automotive Networks: LASAN [MSL+15] is a security protocol that is able to provide multicast and broadcast communication within CAN networks. It uses asymmetric encryption and certification to guarantee authenticity of all components in the network. By managing the communication via a security module, it is also able to arrange communications between ECUs via multicast, thus, providing confidentiality as well. Moreover, by applying further measures during transmission, including timestamps and nonces, also integrity is provided. Lastly, this process is optimized for the use in automotive networks. Hence, it uses symmetric encryption to fulfill the real-time requirements in those networks.

Timed Efficient Stream Loss-Tolerant Authentication: TESLA [PSC+05] is an authentication protocol that was proposed in 2005 to provide fast authentication and integrity mechanisms on the Internet. Nevertheless, due to its features it is also suitable to be used in automotive networks. It is able to provide integrity and authenticity of all packets at the receiver in multicast or broadcast streams. This is done by applying MACs that can be verified at the receiver side only if an unit is allowed to receive this stream. Moreover, it does not need trust between receivers. By applying symmetric cryptography it is able to define message streams that can be verified at each unit individually. This is done by providing the information to verify a packet only to units that are allowed to receive a stream, thus, guaranteeing confidentiality as well [PSC+05].

Transport Layer Security: TLS [DR08] is the successor of the Secure Sockets Layer (SSL) authentication protocol that is widely used on the Internet to enable secure connections (e.g. in Hypertext Transfer Protocol Secure (HTTPS)). This authentication protocol uses asymmetric encryption and certification during the Handshake in order to negotiate and exchange security information. Once both parties agree upon those information, Application data can be sent by applying encryption and compression mechanisms, as well as MACs on all sent messages. It enables authenticity by applying certification and integrity by using MACs. Confidentiality is guaranteed by the Handshake mechanism that is performed for each receiver individually. It provides information to verify and decrypt received messages only to allowed receivers. This protocol is already established on the Internet, but is also suitable to be used in automotive networks. This is because it uses a two step approach, in which in the first step, the Handshake has a high computational complexity due to asymmetric encryption mechanisms. However, the second step, that transmits all Application data, requires less computational power, as mostly symmetric operations and MACs are applied [DR08].

In this section the basic terms of secure networks are introduced. Those include the security components and protection targets. Also, essentials of cryptography including symmetric and asymmetric cryptosystems are discussed. Those are complemented by the introduction of

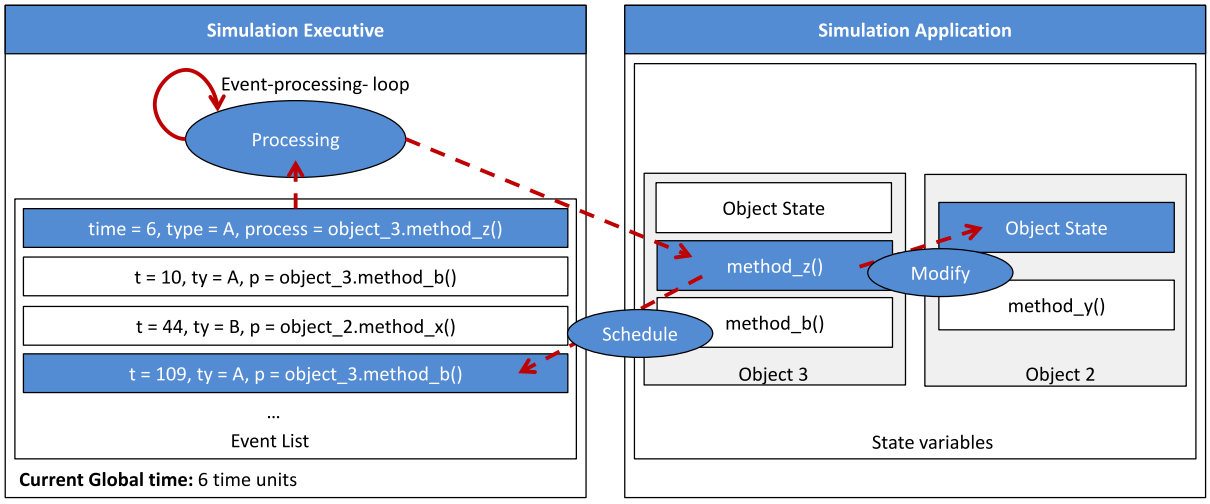

Figure 2.9: Components of a discrete-event simulation. Process of executing a method that is scheduling a new event in the event list and modifying the state of Object 2.

hashing mechanisms and basic principles of authentication, including MACs, digital signatures and certificates. Lastly, the three authentication protocols that are relevant in the scope of this thesis are presented. Those are LASAN, TESLA and TLS.
After the security aspects of this thesis are discussed, next, basics of simulation are introduced, in order to obtain an understanding in which environment the proposed simulator acts.

2.4 Discrete Event Simulation

In the following, discrete-event simulation as an import analysis tool is described. Therefore, first its basic structure is explained, before subsequently, event-driven simulators as a subtype of discrete-event simulators are introduced [Fuj99].

2.4.1 Basic Structure

A simulator according to [Fuj99] is a system that represents the behavior of another system over time. It is executed by a computer program and is emulating the simulation, that is running in the so called physical system. The state of the system is represented using state variables and the simulation dynamic is implemented by changing those variables over the simulation time. In this context the term simulation time refers to an abstract representation, that models the order of events on the physical system. In contrast to that, the wallclock time is the time that actually passes in the real world, when the simulation is run. In discrete-event simulation the simulation time is not related to the wallclock time. Discrete simulation systems are decoupled from the real world's understanding of time. This is because from a computational perspective, the state in the simulation changes every time some impulse is given. In event-driven discrete-event simulation this means, that the state is brought up to date only when events happen in the simulator. Thus, the updating is done in various intervals that depend on the points in the

simulation time when the events occur. This process is described in more detail in Section 2.4.2 [Fuj99].

2.4.2 Event-driven Simulation

According to [Fuj99], in an event-driven discrete-event simulation system (see Figure 2.9) all events are marked with the point in time when they occur in terms of the simulation time. Additionally, the type of event and its effect are saved in the event. The events themselves are stored in a so called event list that is sorted according to their simulation time in ascending order at all times. The current time of the simulation is hereby saved to the global time variable. On top of that, the current system state is stored in the state variables. The global time and the event list are part of the simulation executive that provides the framework of the simulation, whereas the simulation application is characterized over the state variables. In order to execute the simulation all events in the event list are invoked sequentially in the event-loop, thereby manipulating the system state or adding new events (scheduling). Furthermore, the event-loop ensures, that the global time is always updated and old events are removed from the event list. Starting the simulation is done by setting an initial configuration, while stopping it is done by adding a stop event, by defining an end simulation time or by waiting until all events are processed. [Fuj99].
A framework that provides event-driven discrete-event simulation is SimPy [Tea15]. It is implemented in Python and allows to define an environment similar to a parallel processed computer program. In this framework it is possible to define processes that run certain tasks and execute actions that can change the system state. If a process needs time to finish a task, it can wait a defined simulation time before it continues. Next to this virtual parallel processing, SimPy also allows interaction between processes. Hence, one process can interrupt another process or also wait for it by using synchronization objects. A SimPy process is a method that runs in a loop. In this loop a timeout command of t_{wait}, that was called at the simulation time t_{now}, pauses the execution of this method until all events prior to $t = t_{now} + t_{wait}$ have passed. After that, the method is continued at the same code line until another SimPy command is invoked.

In this chapter an overview of essential background information is given. This includes basics about automotive networks including their components, characteristics and functioning. A focus is also put on CAN FD buses as those are the ones implemented in the simulator that is proposed in the scope of this thesis. On top of that, essentials of security and in particular authentication protocols are discussed in this chapter, as those are to be investigated on the introduced simulator. Thus, as the basics are covered, next, the necessity of the introduced simulator is discussed and existing simulators are presented.

3 Related Work

In this chapter a short overview over the literature is given that is related to the implementation of the simulator proposed in this thesis. First, literature is presented that underlines the need for security protocols in internal automotive networks and the urge to investigate their timing behavior. Next, existing simulators that emulate a whole or a part of a CAN network are presented. At the same time their applications are discussed. Also, the applications and features that distinguish the proposed simulator from existing simulations are presented.

3.1 Security in Automotive Networks

Security in automotive networks is an uprising topic. Today's internal automotive networks are insecure and consist of multiple components and subsystems that are connected to each other and to external networks, which provides them with a targets to attackers. Attackers might intend to obtain access to internal networks of the car to steal it, to manipulate the mileage, to do chip tuning or to cause harm to other people [WWW07]. In order to do so they need to access the intra-vehicular communication system and infiltrate the components in the system. According to [SNA+13a] this can be done physically by accessing the On-Board-Diagnostics (OBD) port [KCR+10], the Universal-Serial-Bus (USB) port [WWW07] or the multimedia system [CMK+11]. Also it can be done remotely by exploiting the Bluetooth, Wifi, 3G/4G or Global Positioning System (GPS) connections of automotive networks [CMK+11]. In [KCR+10] it was shown, that having physical access to the network, it is possible to disable breaks or stop the engine on demand, while ignoring the driver input. In [CMK+11] it was presented that this attack can also be performed remotely by uploading custom firmware onto the car's components. This enables the attacker to obtain control over the car's telematics unit as an entry point to control vehicular functions or to exfiltrate the car's location. In [RMM+10] it was shown that

also the Tire Pressure Monitoring System (TPMS) can be attacked by eavesdropping, making it possible to reverse engineer the transmitted messages. This enabled the attackers to endanger the driver's privacy. It is even possible to use the knowledge that is gained about those messages to trigger tire pressure warnings from a distance of up to 40 meters. On top of that, in [FDC11] relay attacks are performed to exploit the Passive Keyless Entry and Start (PKES) system to access the car.
Most of those attacks are possible because of the lack of security measures in today's cars. As the CAN bus is used for safety critical communication methods, mechanisms to prevent those attacks need to be performed on the communicating units on the bus. For this reason, authentication and encrypted communication can be used [WWP04] inside of the networks by applying security protocols [GMVV12, MSL+15] to ensure authenticity, confidentiality and integrity in those systems [SNA+13b]. Nevertheless, those protocols require cryptographic operations that are costly in terms of resources and computational capacity [WWW07]. As those operations are performed on ECUs that are limited in terms of computational capacity, it has to be ensured that the used authentication protocols fulfill the real-time requirements needed in such communication systems [WWW07].

By applying the simulator that is introduced in this thesis, authentication protocols can be validated. This can be done on a temporal basis, to validate if real-time constraints are satisfied within intra-vehicular networks. Also, it can be used to determine the behavior of those systems, when above attacks are introduced in order to optimize protocols for those systems and to evaluate their vulnerability.

3.2 CAN Bus Simulators

In literature CAN networks are simulated diversely. X-In-The-Loop simulations are used to optimize and develop the network or its components. They include the following. Hardware-in-the-Loop (HiL) and Processor-in-the-Loop (PiL) simulations are used to test prototypes of hardware components, while Model-in-the-Loop (MiL) simulators are used to evaluate the risk of algorithms on components and Software-in-the-Loop (SiL) simulations are applied to test the functionality of algorithms [MfSDA14]. Further abstracted software simulators are implemented to perform one of the following tasks. They are used to verify new concepts, to evaluate the performance of an existing network, to verify a network or to configure and test hardware parts of the network. Therefore, the following CAN network implementations were proposed.
For the simulation at hardware level, the tools System Vision [NGM09], CANoe [MNB11], Matlab/Simulink [MNB11] and SIMsystem [App04] are applied in literature. System Vision can be used to design, integrate and verify existent CAN networks on a physical level [NGM09, Men15]. CANoe is used to simulate ECU networks in software to perform both SiL and HiL tests [Vec15]. In [LWWH11] for instance, it is applied in combination with Matlab, in order to analyze the behavior of engine throttle control for the traction control system and in [ZLL+10] to implement a CAN network in order to verify a control strategy for cars. In [App04], SIMsystem is proposed to integrate CANoe to test hardware modules, before inte-

grating them into the network.
To simulate and analyze CAN networks on a software level, the MiL tools prevailing in literature are OMNeT++ [Ope15], NS-2 [DSL+15], NS-3 [NS-15], Matlab/Simulink [MNB11], RTaW-Sim [Rea15], TrueTime [CHL+03], SystemC [DKM10], SpecC [GZD+00] and OPNET [Riv15].
OMNet++ [Ope15] is an object-oriented modular discrete-event network simulation framework, that makes it possible to model any type of network including automotive bus systems. It is based on a modular structure similar to the one presented in this thesis. It is composed of modules that are written in C++ and can be built together flexibly by the user. This makes it also possible to model different protocols for example. On top of that, in [MMT+13] also a CAN system implementation is presented that is based on this simulator.
NS-2 [DSL+15] is also a discrete-event simulator that is written in C++ and can be used to analyze various types of networks, including automotive networks. It has numerous network protocols implemented and is able to simulate traffic, routing or multicast in networks. It can be extended by user-defined protocol implementations enabling a flexible way to evaluate their functionality in various network architectures. NS-3 [NS-15] is also a discrete-event network simulator, which is mainly used to simulate Internet networks. It provides interfaces to enable real-time simulations and supports various protocols on different OSI layers including IP. It provides a simulation core that can be used to analyze wired and wireless networks. RTaW-Sim [Rea15] simulates CAN buses in various standards including CAN 2.0 A and B or CAN FD. It offers various configuration options including the selection of communication protocols or the controller that is used. It aims to help configuring the components in the network in order to improve the system performance and to validate their behavior after integration. This is done by analyzing characteristics of the bus, such as the load level or the worst-case response times of messages. Thus, in [MNB11] it is used to analyze and evaluate the impact of clock drifts in individual network components on a CAN bus. There, it is determined if safety constraints are met on a timing basis if the clock drifts diverge. TrueTime [CHL+03] is a toolbox for Matlab/Simulink that is used to determine transmission delays between nodes and to simulate multi-scheduling algorithms in Networked Control Systems (NCS) [WH13] that apply CAN buses. Nodes are modeled with Simulink and their sending behavior and bus access is handled by TrueTime. SystemC is a library for C++ that uses macros and functions in order to model modules that can communicate with each other via ports and signals, while running in processes. Thus, those simulator are used to simulate and verify CAN buses at a functional level, including the correct data transmission and the arbitration behavior [DKM10]. Also, in [BBT04] the verification of the executable code on CAN network components is performed. This comprises the verification of temporal constraints, buffer behavior and I/O behavior of the network components. SpecC, similarly to SystemC, describes the CAN network on a system level [GZD+00] using macros. It is applied in [SD05] to determine the temporal behavior of CAN networks when multiple nodes are connected to one bus. Three different models of CAN are implemented including functional models and Transaction Level Models with and without arbitration. Lastly, OPNET Modeler [Riv15] is a discrete-event simulator that is used to analyze and design communication networks. In [HWG11] this tool is used to implement the CAN bus

protocol according to its OSI layer definition. As a result, it is shown that the temporal behavior including end-to-end times and data throughput can be analyzed with this framework. On top of that, further abstract CAN bus models are implemented that are tailored to their applications. They include an implementation to improve the scheduling [NK14] or to analyze methods to enhance the timing behavior in CAN networks [ZTS11].
As described in Section 3.1, security mechanisms in automotive networks require real-time behavior in order to provide safe communication between ECUs and most of these tasks are performed on CAN networks. That is why it is necessary to investigate security protocols [GMVV12, MSL$^+$15] on a temporal basis to ensure their suitability in those environments. This can be done using simulations. Most of the simulators described above are focused on the verification of CAN networks by looking into message transmission times between the nodes. However, none is used to specifically focus on the processing times within ECUs based on delays that are introduced by security mechanisms.

In order to validate authentication protocols, a layered representation and a modular simulator structure is advantageous. This is possible to implement in simulators, such as OPNET Modeler [Riv15] or OMNet++ [Ope15]. With this representation it is possible to investigate different protocols of various standards in the same environment, enabling a direct temporal comparison of protocols on all layers. That is why, the proposed IVNS uses a similar approach in order to guarantee a high modularity of components in the simulated automotive network. On top of that, dependencies between parameters and timing values need to be declared in order to be able to compare different authentication protocols with varying algorithms and keys. Some of these simulators, such as e.g. OMNet++ [Ope15], support this creation of customized parameters that can have such dependencies. The proposed IVNS, however, integrates those dependencies and timing values differently as it is focused on security mechanisms in internal automotive networks and on extensibility as well. This means that mechanisms are provided by the simulator that enable easy modeling, validation and comparison of authentication protocols in an intra-vehicular communication system. For instance realistic measurements of security processing times can be applied that define the timing behavior of the simulator and can be extended by the user. This is done via free parameter assignment in the simulation that applies dependencies defined in an external database. Additionally, the proposed IVNS also allows to track all actions that occur in the simulation by using tags, enabling the user to exactly comprehend the conceptual behavior of all components in any scenario implemented. This allows to easily extend, integrate and verify new user-defined protocol implementations.

In this chapter the motivation for the proposed simulator is elucidated. This is done by first, underlining the necessity for the analysis of security protocols in automotive networks and by then, describing existent comparable simulators. With this in mind, in the following chapters the simulator and its components are explained. The first of those components are measurements that were performed on a STM32 microcontroller in order to determine realistic temporal values for the simulator. Those are explained in the next chapter.

4 Measurements of Cryptographic Algorithms

In the simulation model of the IVNS, ECUs communicate with each other by exchanging encrypted messages on a bus. Those messages vary in terms of size and the algorithm that is used for encryption. On the sending side, those messages are encrypted and sent, while on the receiving side they are decrypted and processed. During each encryption and decryption a time passes. Those processing times depend on the processed data sizes and the cryptographic algorithms, together with their parameters. Furthermore, those encryption times depend on the computational power of the device that is used for this operation. In automotive networks the processors executing those tasks are limited in computational power in comparison to computer networks that currently apply those operations in their security protocols.
In order to provide this correlation between processing times and input sizes, measurements were made under varying circumstances. Those records were then input into a database that is accessible to the IVNS. This way, times for cryptographic operations of AES, ECC and RSA were measured. Also, the algorithm characteristics and the input sizes were varied. To achieve similar timing behavior as in todays cars, those measurements were performed on a STM32 microcontroller, which has a similar computational capacity as state-of-the-art ECUs.
In the following, the environment in which the measurements were taken is shown in Section 4.1. Afterwards, the implementation is described in Section 4.2, before lastly, in Section 4.3, measurements are verified for correctness.

4.1 Environment

The measurement environment consists of the Mini-M4 evaluation board and the IAR Embedded Workbench connected to it. It is complemented by the CyaSSL and the STM32 Cryptographic Library as the cryptographic frameworks whose performance is evaluated on the board.

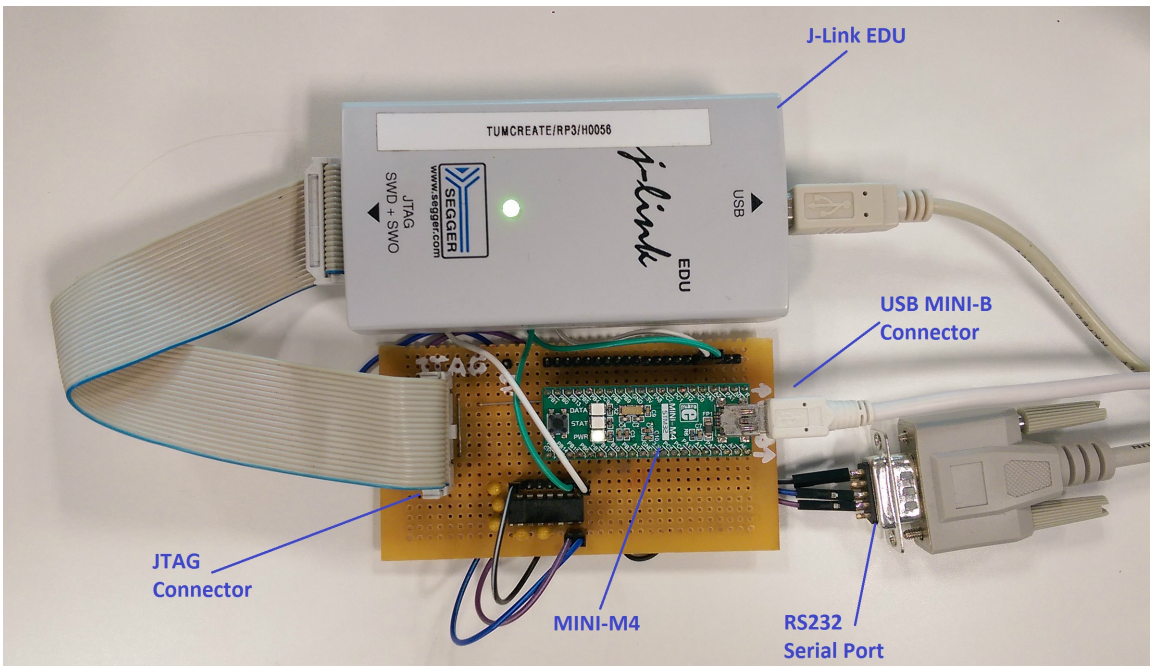

Figure 4.1: Evaluation Board used for timing measurements.

4.1.1 Evaluation Board

Today's cars include various microcontrollers, consisting of microprocessors, flash memory, programmable read-only memory (PROM), I/O ports, signal generators and timers. Common microcontrollers [XMO13] are Infineon's TriCore microcontrollers, Atmel's AVR family or upcoming microcontrollers such as XMOS XS1-L16A-128-QF124 [Aga15]. Concerning the specifications, less complex applications can be performed on an Atmel's XMEGA AVR, that has a 8-Bit AVR architecture with program memory of up to 384 KB and clock speeds of up to 32 MHz [Atm15]. For more complex tasks, XMOS XS1-L16A-128-QF124 can be used [XMO13]. This 16-core microcontroller is capable of executing 1000 Million Instructions Per Second (MIPS) shared between 16 real-time logical cores each processing 32-bit instructions. As those components vary in performance, for the measurements performed in this thesis a microcontroller is used that lies in the middle of current microcontrollers in terms of performance. Thus, the STM32F415RG microcontroller by ST-Microelectronics is used for evaluation [ST 15]. It comprises a 32-bit CPU running on frequencies of up to 168 MHz and having 1 MB of flash memory. On top of that, it is suited to perform cryptographic operations as it has built-in acceleration for AES, hash, random number generation and HMAC. Choosing a microcontroller like this gives realistic results for simulations of automotive networks as ECUs are getting more powerful and hardware acceleration is getting more common. This microcontroller is installed on a MINI-M4 development board and connected over a JTAG connector to a Segger J-Link-EDU that is used to flash the device via the connected computer (see Figure 4.1). Power supply is provided via a USB MINI-B connector on the MINI-M4. In order to read the results, this board is linked to a computer via an RS-232 serial port. By writing to the pins of this port, output is read out on the PC side using a serial terminal software, such as HTerm.

4.1.2 Development Environment

The integrated development environment IAR Embedded Workbench [IAR15] is used to program and debug the microcontroller via the connected J-link.
Based on this environment two encryption libraries are evaluated. Those are the STM32 Cryptographic Library [ST 13] by ST-Microelectronics and CyaSSL [Wol14, Wol15b]. The former library enables cryptographic operations in two ways. It is able to run either with or without integrated hardware accelerators for cryptography, which support AES modes, hashing methods and random number generation. In software, this library includes AES with various key lengths in ECB, CBC, CTR, CCM, GCM or CMAC mode. Also the hashing methods MD5, SHA1 and SHA256 are supported. Moreover, it is also possible to sign and verify bytes using RSA signature functions with PKCS-1 or the ECC variant ECDSA. Lastly, ECC key generation and random number generation based on DRBG-AES-128 are available.
The other library that is used is CyaSSL, which is now known under the name WolfSSL. CyaSSL is a lightweight C-language based framework for embedded devices, that is powered by the wolfCrypt Library and is FIPS 140-2 validated. It supports AES with diverse key lengths in CBC, CTR and CCM mode, as well as signing and verification over RSA or ECC with ECDSA. Moreover also MD5, SHA1 and SHA256 hashing is available. Furthermore, keys can be generated for RSA and ECC. Lastly, ECC can be used for encryption and decryption.

4.2 Realization

The measurements are performed by implementing an algorithm that measures processing times and outputs the results over the RS-232 interface. The resulting timings are then stored in a database that is used in the simulator. Those steps are realized as follows.

Timer specification: In order to determine the encryption times, the duration of certain lines of code need to be measured. Thus, in front of the relevant lines a timer is started and after their end it is stopped. For this reason timer TIM7 of the STM32F415RG is used [ST 15]. It counts the cycles that pass between its start and end point at an oscillation rate of 16 MHz. A counter resolution of 16-bit and a prescaler factor are used, that can be set to any value between 1 and 65536. This allows to measure time values between 0.0000000625 and 268.43136 seconds. Hereby, the transformation from counter cycles to computation time is done via

$$time = prescaler * number_{cycles} * (1/rate_{timer}). \tag{4.1}$$

Measurement algorithm: Using this timer, measurements over a set of input lengths are performed on the STM32F415RG. This is done as follows.
First, a random byte string of length 64 is generated that is used as a template. Next, it is defined which input lengths are measured by setting the minimal and the maximal text length that is to be measured. Also the number of runs that is made per text length is indicated.

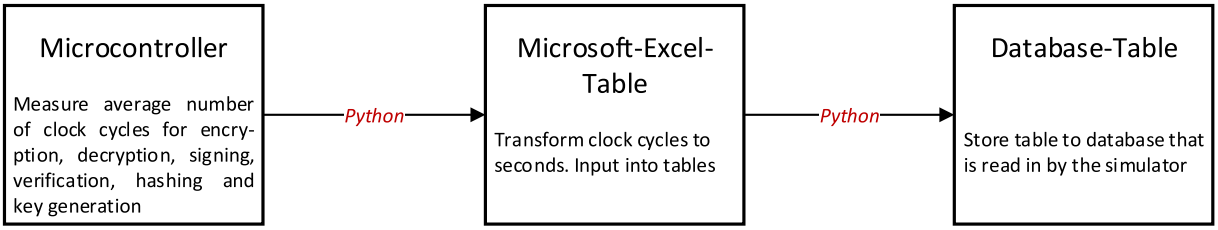

Figure 4.2: Processing steps of performed measurements to determine timing behavior of cryptographic operations on a STM32 microcontroller.

By iterating over all text lengths and by repeating the measurements for the given number of runs, the average encryption and decryption times are calculated. Therefore on each run, per text length, a byte string of the currently iterated length is created from the template bytes. This string is then input into a certain encryption, hashing or signing function that is surrounded by the timer start sequence and the timer end sequence. In that way, the processing times of that operation for a particular input byte length are measured. Consequently, by taking the average of all those runs, per text length, an representative value for processing times of each operation type is found. The output of the measured cryptographic operation is then used to measure the timings of the corresponding decryption or verification methods in the same manner. The key generation is measured similarly, by using the corresponding library methods and by repeating the key generation for the defined number of runs.
For each of those measurement groups, certain parameters are varied. Those are the algorithms MD5, SHA1, SHA256, AES, RSA and ECC, together with their parameters and the used cryptographic library.
For AES diverse key lengths and modes are investigated. In AES the evaluation is performed with or without hardware acceleration. In RSA the key length and the exponents are varied, while in ECC parameter lengths are diversified. The results of those experiments are presented in Section 8.1.

Processing steps: After each evaluated text length the average number of passed cycles and the applied text length itself is output via the RS-232 interface and read out using HTerm. By execution of a Python script, this data is then transformed into the unit seconds by applying Equation 4.1. At the same time, this data is stored to a Microsoft Excel table that is used for the manual validation of the measured data. Once the data is validated, it is further input into the measurement database that is used by the simulator. This step is also performed via a Python script that automatically parses the tables and puts its entries into the database. An overview of those steps is given in Figure 4.2.

4.3 Validation

All measurements were validated afterwards. This is done in three ways.

Matlab plots: By plotting the measured size values on the abscissa and the processing time on the ordinate, all measurements are investigated and compared. For this purpose MATLAB by MathWorks is used as a plotting tool. Next, inspecting the curves leads to the debunking of outliers. Unmasked errors are then measured anew.

Plausibility checks: On top of that, plausibility of measurements was checked according to the following deductions.

- All algorithms perform better on the STM32 Cryptographic Library than on the CyaSSL library as the former is published by the producer of the used microcontroller.
- Hardware implementations run faster than there software counterpart.
- Symmetric algorithms and hashing methods require less computational operations than the asymmetric algorithms RSA and ECC. Thus, the former run faster.
- RSA encryption needs more processing steps than its decryption and thus, it is slower.
- RSA pads input data to the key length before processing it. Thus, RSA times are independent from the input data length.
- For AES, RSA and ECC a longer key length leads to more calculation steps. Therefore, longer keys take a longer time.
- For AES and hashing, a longer text length means more calculations as well, resulting in longer processing times.

Benchmark comparison: In addition to those deductions, the resulting timings are compared to benchmarks provided by the developers of both libraries and are thus, inspected to be plausible. The corresponding benchmarks are given in [Wol15a, ST 13]. For CyaSSL those benchmarks are performed on evaluation boards [Wol15a] that are different to the ones in automotive applications. On those units the relations between times are used to check plausibility. Above that, for the STM Cryptographic Library benchmarks are performed on the same microcontroller as the one used in this thesis [ST 13]. Thus, a direct comparison is possible. Nevertheless, both libraries only provide a low number of measurements. Those do not comprise the full extent of measurements needed to simulate the behavior for all input sizes that are needed within the security protocols of the simulator. Thus, measurements had to be taken in the scope of this thesis.

In this chapter, it is described how measurements of cryptographic operations were performed and validated. Those values are the basis for realistic simulations in the IVNS that is proposed in this thesis, as those are used to determine processing times in it.
After the basics and the STM32 measurements are explained, in the next chapter the implementation of the simulator and all its components are presented.

5 Implementation

In this chapter an extensible and modular discrete-event simulator (IVNS) is proposed as a tool for the analysis of conceptual, temporal and security behavior of automotive networks. In particular, it allows to analyze security protocols on a temporal and conceptual basis. It provides tools to run simulations with various network architectures, as well as network devices with diverse hardware and software modules. This flexibility is enhanced by user-defined network components and their configurability. On top of that, information extraction is implemented customizable, enabling user-defined output depths. Furthermore, data of the simulator may depend on measurements that are dependent on other parameters. Thus, an extensible database is implemented that allows to control the timing behavior of components based on input parameters. To achieve realistic results, those dependencies were measured on a STM32 microcontroller, as described in Chapter 4. Visualization of data is enabled by a GUI extension for the API. To ensure correct behavior, the simulator was validated with state-of-the-art software testing methods, including unit and integration tests. Diverse simulation results can be generated with a test case generator implemented for this simulator. Lastly, an Adapter was implemented that connects to the existing BMS simulator proposed in [Mee15], enabling an analysis in a realistic application case. An overview of those subareas is given in Figure 5.1. The simulation is described in this chapter, whereas the testing mechanisms are detailed in Chapter 6 and the battery management integration in Chapter 7.

The presented simulation is implemented in Python and SimPy, with the Eclipse IDE as the development environment. It is designed according to the Model-View-Controller (MVC) pattern [KP88] that comprises the components illustrated in Figure 5.1. The components of this pattern are introduced sequentially in this chapter. Those are the following. The MVC pattern consists of the simulator core that is introduced in Section 5.1 and forms the model component in this pattern. Additionally, the controller according to the MVC pattern is composed of the

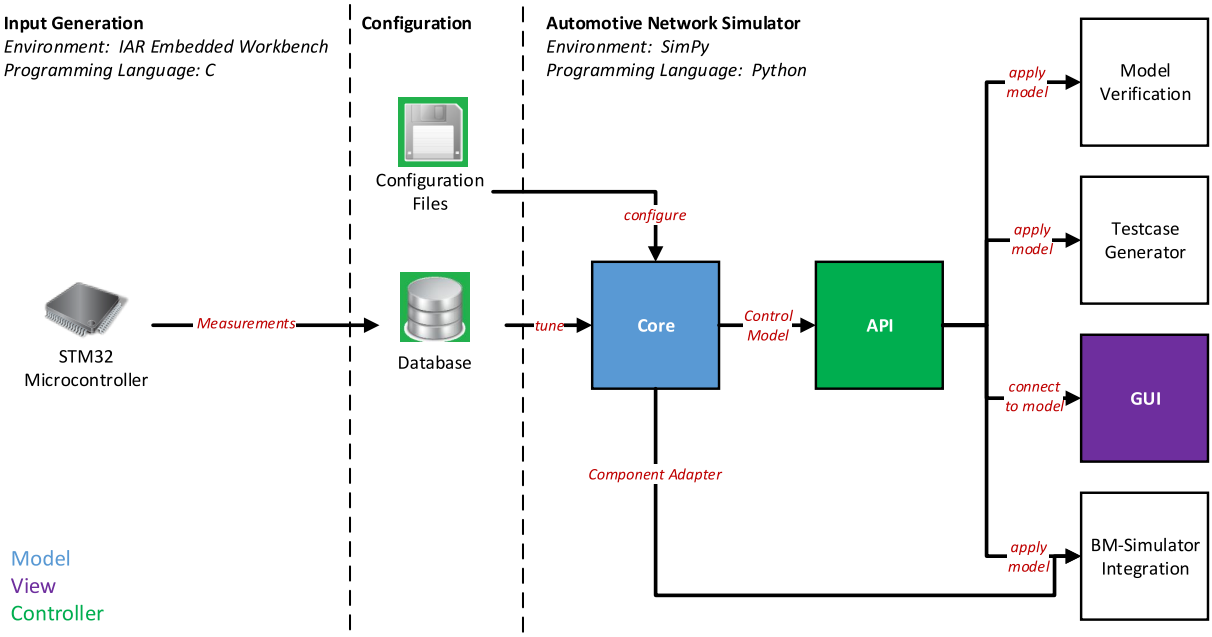

Figure 5.1: Overview of all implemented components of the proposed simulator (IVNS).

configuration, the API and the Input-Output processing of this simulator. Those are explained in Section 5.3, Section 5.4 and Section 5.5. Lastly, the view of this pattern is implemented as the GUI and is explained in Section 5.6.

5.1 Core

The core of the simulator is composed of the component models, the communication of those components and the security modules that are applied by them.
The component models comprise models of ECUs, buses, gateways and messages, which each run several SimPy processes that simulate their temporal behavior. This behavior depends on the configuration of those components including the choice of their protocols on each layer. Those components are introduced in Section 5.1.1. The communication of those components is enabled by executing SimPy processes that exchange data both within the modeled ECUs and gateways, as well as on connected CAN bus implementations. Those interactions are explained in Section 5.1.2. On top of that, the system includes methods to simulate security mechanisms such as certification, signing or encryption, which are discussed in Section 5.1.3.

5.1.1 Components

In the scope of this thesis solely CAN and CAN FD buses are implemented. That is why all components and concepts presented in the following refer to those network types. In the following the three basic components that form the simulator are discussed. Those are the automotive environment, buses, ECUs, and messages, which are illustrated on Figure 5.2.

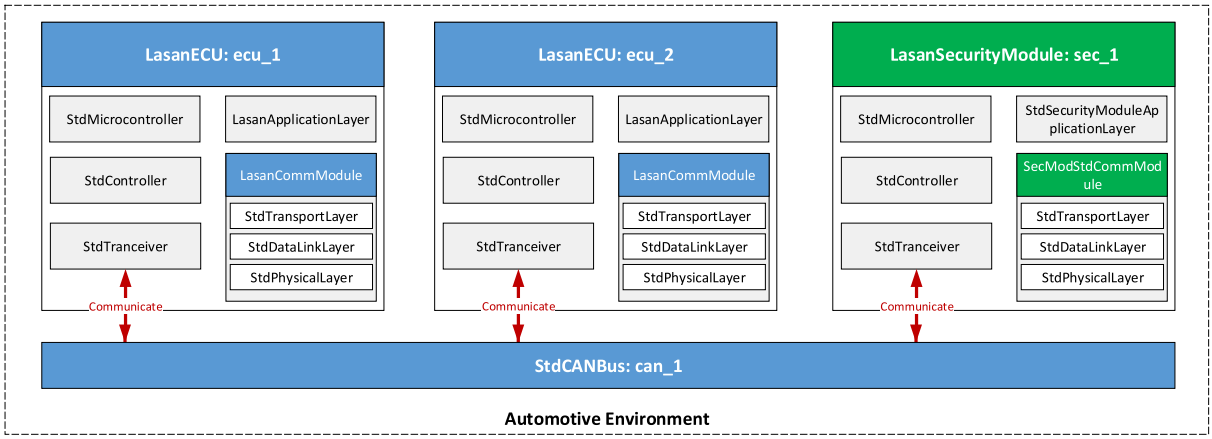

Figure 5.2: Example: LASAN ECUs ecu_1 and ecu_2, as well as security module sec_1 connected to bus can_1 in the automotive environment.

Automotive environment: The automotive environment contains all network components, their architecture and the SimPy environment that manages their discrete-event simulation processes. During setup all components (e.g. ECUs, buses, gateways) are added to this environment. It is used to initialize and start all SimPy processes that run the simulation.

CAN buses: The CAN bus forms the transmission medium of the communication system. It is connected to ECUs, that communicate with each other, and gateways, that allow to transmit messages to other subnetworks via this object. In an automotive environment several CAN buses may exist, that are interconnected arbitrarily. To enable the selection between simulator performance and precision, two CAN bus implementations were implemented, which are introduced in Section 5.1.2.

ECUs: ECUs are modeled in a modular way in the simulation as it is shown in Figure 5.5. Those components contain hardware and software components. In CAN networks, the former consist of a microcontroller, a CAN controller and a transceiver, while the latter comprises an Application Layer and a Communication Module. This module contains a Transport Layer, a Data Link Layer and a Physical Layer. The Application Layer runs applications that send or receive messages. The Communication Module implements security protocols, while the Transport Layer performs segmentation. Via the Data Link Layer, together with the Physical Layer and the transceiver connected to it, frames are put on or received from the bus. Each ECU can implement those layers differently.

CAN bus gateways: A gateway is realized as an implementation of an ECU. In contrast to other ECUs, however, it consists of multiple ports, that each comprise an own Transport, Data Link and Physical Layer, whereby each port is connected to one bus. Incoming messages pass the corresponding three Layers associated with the port. This port pushes them to the remaining ports using their dedicated layer structure. A defined processing time is timed out before the messages are forwarded to the other buses.

CAN and CAN FD Messages: Messages, that are sent by a CAN bus, contain the header and data fields of CAN (FD) frames as described in Section 2.2.2 and Section 2.2.3. Dependant on the selected implementation, CAN or CAN FD frames are sent on the buses.

5.1.2 Execution

The components that are introduced in Section 5.1.1 are setup in order to initialize the simulation which is explained in the first part of this section. In the second part, the way they communicate with each other is discussed.

System Setup: To setup the simulation, all components are created, with the desired configuration, via the API (see Section 5.4). Next, their architecture is defined and a Monitor is instantiated (see Section 5.5). Then, the automotive environment sets up the simulation by starting SimPy processes (see Section 2.4.2) for each ECU, bus and gateway instance.

Timing aspects: To achieve realistic behavior in all started processes, the timing behavior of components is adapted by defining processing times in each of them. For example, an ECU's Application Layer could start a sending process in the Communication Module, followed by a waiting time of two seconds, before the next sending is initialized. Also, on the Communication Layer, some security mechanisms could require processing times to encrypt messages.
In this model it is assumed that the Transport, Data Link and the Physical Layer do not include any processing times. Only times that are triggered by the sending and receiving methods of the Application Layer, the transmission times on the bus, delays in the gateway and the cryptographic methods of the Communication Module are considered. They are the decisive factor for the system's temporal and conceptual behavior and vary along the system configuration.

Intra-ECU transmission: In an ECU, further SimPy processes for the sending and receiving of messages are started from the main process. As it is shown in Figure 5.3, sending of messages is initialized by the Application Layer, which passes them down to the next lower layers until it reaches the transmit buffer. In the same way received messages are pulled from the receiving buffer and pushed up to the Application Layer through those layers. During those sets of processes each layer can introduce processing times if required.

Inter-ECU transmission: Message sending between ECUs is performed via connected buses by pulling messages from the transmit buffer of ECUs and writing them to receive buffers of ECUs. This is implemented in two ways: The Standard and Rapid mode.
In Standard mode, as it is shown on Figure 5.4, each Data Link Layer has a SimPy process method running. As long as there are messages in the transmit buffer, this SimPy process permanently tries to send the highest prioritized message in it. When sending a message, the ECU's Data Link Layer checks if the bus is free. If this is not the case, the Data Link Layer waits until the bus is free again, waits a back-off time and the procedure begins anew. If the bus is free, the Physical Layer pushes its message to the bus. If multiple ECUs are willing to send at the same

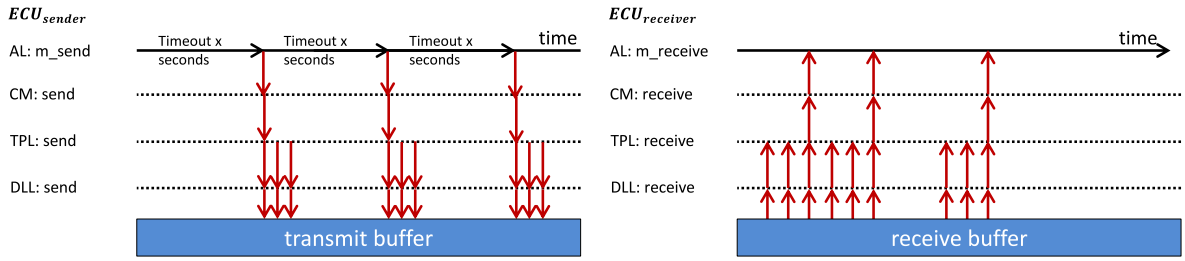

Figure 5.3: Sending and receiving processes from Application Layer to Data Link Layer buffers. In this example, at Transport Layer each message is segmented into three frames. *(AL: Application Layer, CM: Communication Module, TPL: Transport Layer: DLL: Data link Layer)*

time, the bus determines the one with the highest prioritized message and sends its message. The corresponding sender is notified that it was allowed to send, while other ECUs are notified that their request failed. The successful message is then sent by the bus process, that waits for a transmission time and writes the message to the receiving buffers of all connected ECUs. Thus, in the Standard mode messages are actively put on the bus by the ECUs.
As opposed to this, in the Rapid mode (see Figure 5.4) ECUs are passive and messages are pulled from the ECUs by the CAN bus. For this reason the transmit buffers of the ECUs are implemented as priority queues, where the first message has the highest priority. Here, the bus object holds references of all Data Link Layers that are willing to send. Then, the bus sequentially pulls the highest prioritized messages from those Layers' queues and sends them sequentially. Similar to the Standard mode this is done by waiting a transmission delay and by writing the messages to the receivers' buffers. Thus, in the Rapid mode the bus is the active component that puts messages on the bus, while in the Standard mode the ECUs request and put messages on the bus. The Rapid mode is faster, as less SimPy processes are involved.

So far the individual components and their interconnections are introduced. Nevertheless, in order to enable secure communication according to defined authentication protocols, security mechanism need to be simulated. This is described in the next section.

5.1.3 Security Mechanisms

Simulating security mechanisms needs cryptographic tools for the conceptual analysis of secure automotive networks. For this purpose the following security components are implemented.

Encryption, Decryption, Hashing, Compression: Encryption and Decryption are modeled in an abstract way. For this purpose messages are simply wrapped in a Wrapper object together with the key ID needed for decryption. This indicates that encryption was applied to it. It can be decrypted with the key that holds the corresponding key ID. For hashing and compression also messages are simply wrapped. Thus, those operations are also implemented abstractly.

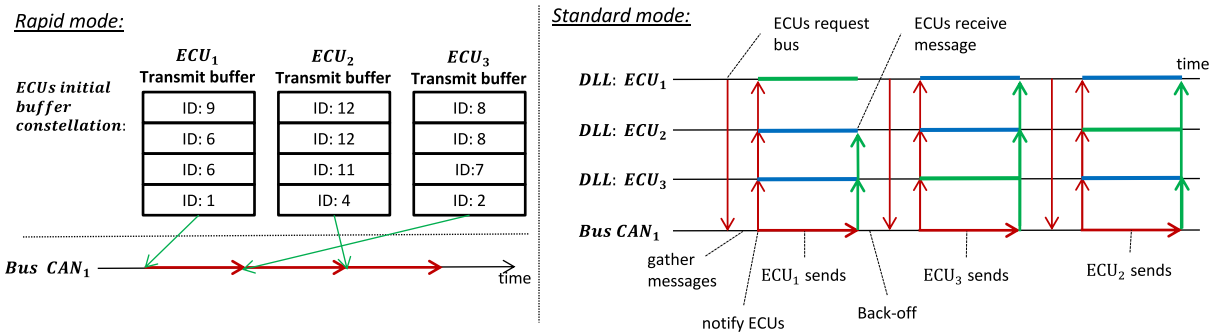

Figure 5.4: Standard bus and Rapid bus communication of three ECUs. Both ECUs have the buffer constellation indicated on left side. In the given situation in both modes ECU_1 sends first, then ECU_2 and lastly ECU_3. (green: sending ECU, blue: receiving ECU)

Certification: A CA hierarchy is implemented and applied to define the structure of certification authorities. This CA Hierarchy consists of CA Nodes that correspond to CA objects. By calling a CA's request_certificate method, an ECU Certificate is generated, that can be validated with the corresponding root certificates of this hierarchy. In order to obtain those certificates, the Certificate Manager is instantiated with a given CA Hierarchy. By providing the corresponding CA ID, the ECU Certificate of this CA and all root certificates needed to verify it, are returned. ECU Certificates are modeled according to the X-509 standard that is described in Section 2.3.3. To verify the validity of a certificate, all root certificates and the current simulation time have to be provided.

5.2 Model Implementations

Based on the core that is introduced in Section 5.1, in this section implementations of the proposed simulation model are described. They include implementations of the Transport Layer and the Communication Layer as they are shown in Figure 5.5 and are explained below.

5.2.1 Transport Layer

Two variants of the Transport Layer are realized. Those are the Standard and the Segmentation Transport Layer. In the standard version, messages, that are incoming from the Communication Module, are wrapped into a CAN or a CAN FD message with arbitrary size and directly forwarded to the bus. This includes in particular messages that are bigger than a CAN or CAN FD frame. Thus, no segmentation is used in this implementation.
In contrast to that, the Segmentation Transport Layer segments messages according to the ISO 15765-2 standard presented in Section 2.2.2. Thus, a message, that is incoming from the Communication Module is segmented into the defined first and Consecutive Frames. The First Frame wraps the actual message that is to be sent and all further messages are empty messages of the defined length for Consecutive Frames. Those messages are then put into the transmit buffer.

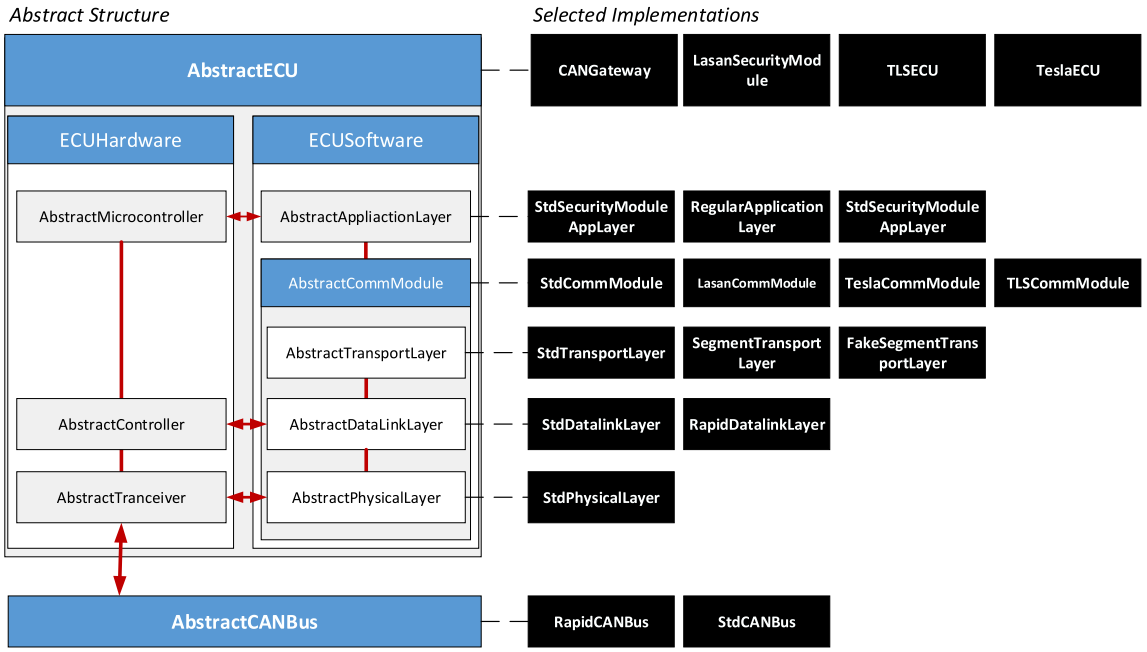

Figure 5.5: Basic structure of ECUs and CAN buses and selected implementations.

Moreover, in case that CAN FD is used, the data fields are padded to the allowed sizes (see Section 2.2.3). On the receiver side, the actual data is reassembled and passed to the Communication Module. This is shown schematically in Figure 5.3. For all analyses made in the scope of this thesis, only the Segmentation Transport Layer implementation is applied.

5.2.2 Protocol definitions

Next, three authentication protocols are implemented on the IVNS. In order to better understand these implementations, in the following a short overview of those protocols is given. Those protocols are LASAN, TESLA and TLS.

Lightweight Authentication for Secure Automotive Networks: LASAN [MSL+15] uses a Security Module (SM) that is connected to the bus system and serves as a trusted authority, that administers secure connections between ECUs. That is why it stores a list with all legit connections between them. To establish a secure connection first *ECU Authentication* and then, *Stream Authorization* is performed.
In the *ECU Authentication* phase, ECUs in the network authenticate towards the SM and exchange a symmetric key K_{ECU_i} with it. For this purpose, a *Security Module Advertisement*, containing a certificate with its public key $K_{SEC_{pub}}$, is broadcasted by the SM. This is verified at the ECUs' sides. If this was successful, a Registration Message M_r is responded. It contains a ECU certificate f_e, the SM ID y, a symmetric ECU key K_{ECU_i}, a timestamp t_e and a nonce n. This is sent as $M_r = (E((y, K_{ECU_i}, t_e, n), K_{SEC_{pub}}), E(h(y, K_{ECU_i}, t_e, n), K_{ECUpriv,i}), f_e)$, where h is a hashing function and $K_{ECUpriv,i}$ is the private key of the sending ECU. Once received by the SM this message is verified. If this was successful, the ECU key K_{ECU_i} is stored

and a *Confirmation Message* M_c is answered to the according ECU with ID e_i. It is defined as $M_c = E((e_i, n, t_e), K_{ECU_i})$.
After an ECU received a Confirmation Message, it is allowed to request streams during the *Stream Authorization*. If an ECU wants to send a stream with ID s_k, it requests permission from the SM to be able to send secure messages within this stream. The SM can sent point-to-point messages to ECUs by sending them encrypted with the respective symmetric ECU key K_{ECU_i} of the target ECU.
The request starts with a *Request Message* $M_{rq} = (s_k, e_i, t_e, n)$, that is sent by the requesting ECU with ID e_i. If a legit message stream was requested, the SM generates a session key K_{sess_k} for this stream. This key is forwarded to permitted receiving ECUs and the sending ECU via the *Grant Message* M_g that comprises the sender ID e_s and the receiving ECU ID $e_{r,i}$. It is formed as $M_g = (e_s, e_{r,i}, s_k, K_{sess_k}, n, t_e)$. If a stream is not granted, a *Deny Message* is sent which contains the same content as a *Grant Message* excluding the session key. By encrypting messages with the session key K_{sess_k} the sender e_s is able to transmit messages via multicast directly to all receivers $e_{r,i}$ with this key.

Timed Efficient Stream Loss-Tolerant Authentication: TESLA [PSC+05] uses MAC verification to ensure a secure communication. This is done by sending messages, that contain a verification hash that was hashed using a certain MAC key and the MAC key of a prior hashing. Thus, the receiver stores incoming messages and verifies them once it receives the corresponding MAC key.
The process is divided into *Time synchronization*, *Sender setup* and *Message Exchange*.
First, in *Time synchronization*, all senders' clocks are synchronized to their receivers' clocks. This is done by exchanging the sender's time t_s with the receiver's time t_r, resulting in sending delay $D_t = t_s - t_r + S$, that is stored by the receivers. S is the clock drift of the session.
During the *Sender setup*, per stream s_k that a sender is willing to send, the sender divides its future time uniformly into intervals of duration $t_{int,k}$, while at the same time assigning one MAC key to each interval. For this purpose N keys are generated. Then, the first key K_N is initialized with a random number. Using a Pseudo Random Function (PRF) keys K_0 to K_{N-1} are derived from it. The key K_i is determined from key K_{i+1} by calculating $K_i = F(K_{i+1})$, where F is a PRF. Moreover, the first key K_0 of this chain is sent to all receivers of the stream using asymmetric or symmetric encryption. On top of that, both parties agree upon a further PRF $F'(K)$.
Next, messages can be broadcasted and received during *Message Exchange*. To send messages the sender determines the key K'_i from key K_i corresponding to the current time interval i using $K'_i = F'(K_i)$. Also it prepares the key K_{i-d} of the time interval $i - d$. Then, the packet $P_j = (M_j, i, MAC(K'_i, M_j), K_{i-d})$ is broadcasted. Then, at the receiving side safe messages are stored in a buffer and read once they are verifiable. When a packet P_j is received, the safe packet test verifies that this packet is secure and stores it in the buffer. Next, the received key K_{i-d} is verified by using previous keys. Lastly, all packets P_h with content M_h in the buffer, that were sent in interval $i - d$, are inspected. For this reason $K'_{i-d} = F'(K_{i-d})$ is calculated from the received key K_{i-d}. Subsequently, the $MAC(K'_{i-d}, M_h)$ is determined and verified

against the hash contained in packet P_h. If all of those verifications are successful, the message is verified and passed to higher Communication Layers.

Transport Layer Security: TLS [DR08] consists of two sublayers. On the lower Record Layer segmentation, compression, MACs and encryption are applied according to the defined cipher suite. However, the upper sublayer is used to establish a secure connection using the Handshake once per stream with ID s_k. During this phase, a cipher suite is negotiated. It specifies encryption, PRF, compression and MAC algorithms, that are used to read or write data, as well as its keys. It works as follows.
Clients initialize a stream by sending a *Client-Hello* message, that comprises among others a random byte string, the session ID and a list of cipher-suites that the client supports.
After receiving this message the server sends a *Server-Hello* message with a random byte string and a cipher suite that is supported by both parties and that is to be used for securing this message stream. Next, a *Server-Certificate* with the server certificate and all root certificates needed to verify it are encrypted and sent. This is followed by a *Server-Key-Exchange* message, that contains information needed to exchange the pre-master-secret. Then, a *Certificate-Request* message is sent that demands the certificate of the receiver. Lastly, an empty message called the *Server-Hello-Done* message finishes the end of the Server-Hello process.
The client verifies the prior messages. Next, it sends the *Client-Certificate*, followed by the *Client-Key-Exchange* message. The latter sets the pre-master-secret, that is encrypted using the public key of the server and then sent. Consequently, in the *Certificate-Verify* message the client signs all messages that were cached so far using negotiated algorithms and transmits them. In this step the master secret is calculated using the exchanged random numbers and the pre-master-secret. The master secret is used to generate MAC keys and encryption keys for the requested stream. Next, a *Change-Cipher-Spec* message is sent, that demands the client to apply the negotiated cipher-suites for the further communication. Finally, the *Finished* message is sent and encrypted with the negotiate cipher-suite and its keys that result from the master secret. This message also includes a hash generated from all messages that were exchanged so far. It is used by the server to verify the communication.
Once the server receives the client's messages, it verifies them and generates the master secret from the received data analogously, as it was done on the client side. If the verification is successful, again a *Change-Cipher-Spec* message is sent that demands the client to apply the negotiated cipher-suites for the further communication. Lastly, a *Finished* message is sent that contains a verification hash generated from the messages sent so far. If the *Finished* message was successfully verified, Application data is exchanged and the negotiated cipher-suite together with keys from the master secret are used to secure the connection on the Record Layer.

5.2.3 Lightweight Authentication for Secure Automotive Networks

The security protocols LASAN, TESLA and TLS, as they are shown in Section 5.2.2, were each implemented as a Communication Module. Thus, Section 5.2.3, Section 5.2.4 and Section 5.2.5 give an insight about the way communication is realized in the proposed simulator.

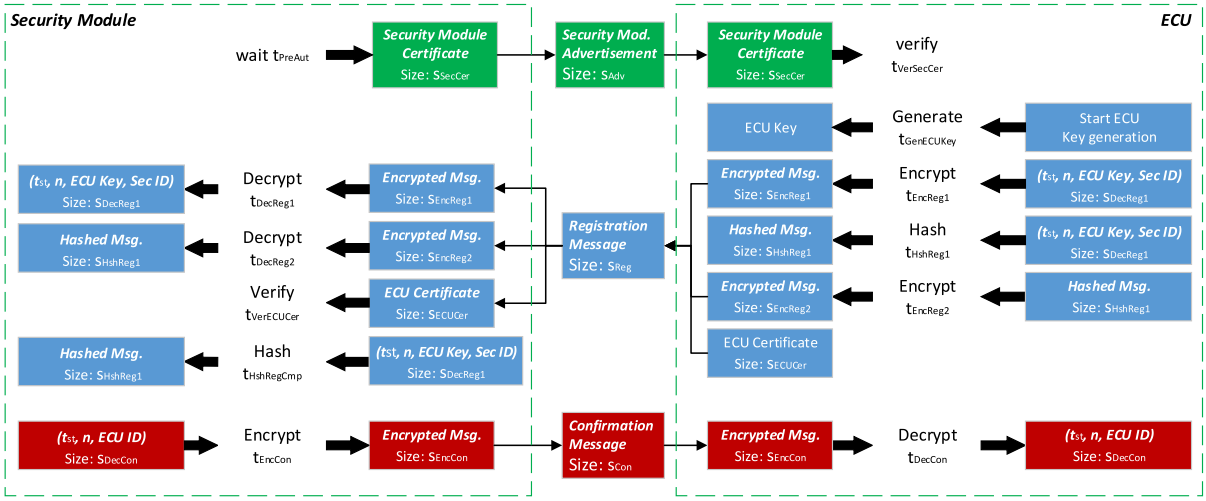

Figure 5.6: ECU Authentication: Timing parameters and sizes parameters that are used in the implementation of this protocol.

During those sections the applied sizes and times are introduced. This will help to comprehend the way those parameters are linked. It will also help to get an understanding of how the configuration affects those parameters, which is described in Section 5.3.3.
Usually ECUs in a CAN (FD) network send messages with certain message IDs in regular time intervals. Each of the ECUs can send multiple message IDs. In terms of the simulator, this means that every time an ECU is willing to send a message, it starts a process in the Communication Module, that transmits the message by applying the defined security protocol mechanisms. Hence, if the Communication Module process is considered in isolation, it can be seen as a system with messages coming in a send method and going out to a receive method. The receive method ensures, that only authorized messages are let pass to the Application Layer upon reception. Also the send method guarantees that all security mechanisms are induced and only authorized recipients are able to read the message. In this consideration it is assumed, that senders and receivers of a message stream are connected to the same bus and thus, are able to exchange messages. In the following, sending and receiving processes of the LASAN protocol as they are implemented are explained.

To simulate LASAN, ECUs and a SM are created. Those communicate via ECU Authentication and Stream Authorization, which are explained below and illustrated in Figure 5.6 and Figure 5.7. The ECU Authentication works as follows.

Security Module Advertisement: After a timeout t_{PreAut} the Advertisement message containing a certificate is sent by the SM. This certificate is generated as described in Section 5.1. The size of the sent message s_{Adv} is the size of the certificate s_{SecCer}.
ECUs, that receive this message at the Communication Module, time out for a certificate verification time $t_{VerSecCer}$. If the verification is successful, a Registration Message is sent.

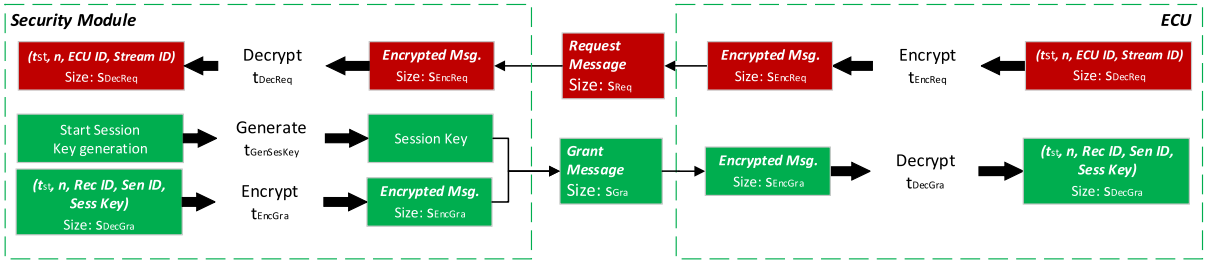

Figure 5.7: Stream Authorization: Timing parameters and sizes parameters that are used in the implementation of this protocol

Registration Message: To create the Registration Message, a timeout is executed for the creation of the symmetric ECU key $t_{GenECUKey}$, for the encryption of the first part of the Registration Message with this key $t_{EncReg1}$, for the hashing of this part $t_{HshReg1}$ and for the encryption of this hashed object $t_{EncReg2}$. The first plaintext is of size $s_{DecReg1}$, while its cipher is of size $s_{EncReg1}$. Analogously, the hashed part is of size $s_{HshReg1}$ after hashing and the second part of the Registration Message of sizes $s_{DecReg2}$ and $s_{EncReg2}$ respectively. An ECU certificate is of size s_{ECUCer}. Thus, the total size of a sent Registration Message is
$s_{Reg} = s_{ECUCer} + s_{EncReg1} + s_{EncReg2}$.
For each ECU, the SM receives a Registration Message. Thus, per arrived message of this type the following timeouts are applied. Those are the decryption time of the first part $t_{DecReg1}$, the decryption time of the second part $t_{DecReg2}$, the verification time for the ECU certificate $t_{VerECUCer}$ and the time to hash a comparison hash for the Registration Message $t_{HshRegCmp}$.
If this message is verified a Confirmation Message is sent.

Confirmation Message: This message is sent to each authenticated ECU. Before it is sent, however, a timeout of t_{EncCon} for the symmetric encryption of this message is applied. The size of the message is s_{DecCon} before and s_{EncCon} after encryption. The sending size is s_{Con}.
Once the ECU receives the Confirmation Message, it times out for the decryption time t_{DecCon} needed for the symmetric decryption of this message. This allows it to communicate with other ECUs after the Stream Authorization.

After this phase is finished, communication between ECUs is possible. The Stream Authorization process is shown on Figure 5.7 and implemented in the proposed simulator as follows.

Stream Request Message: A Request Message introduces a symmetric encryption delay t_{EncReq} needed to encrypt the message, that is then sent. The clear message size is s_{DecReq} and its cipher size is s_{EncReq}. It is equal to the size s_{Req} of the sent message.
The SM receives the request and decrypts it symmetrically, while applying a timeout of t_{DecReq}.

Grant / Deny Message: If an allowed stream is found for this request, Grant Messages are sent to all receivers. Else a Deny Message is sent to the requesting unit. For the Grant Message, a timeout $t_{GenSesKey}$ is applied for the generation of a symmetric session key and per

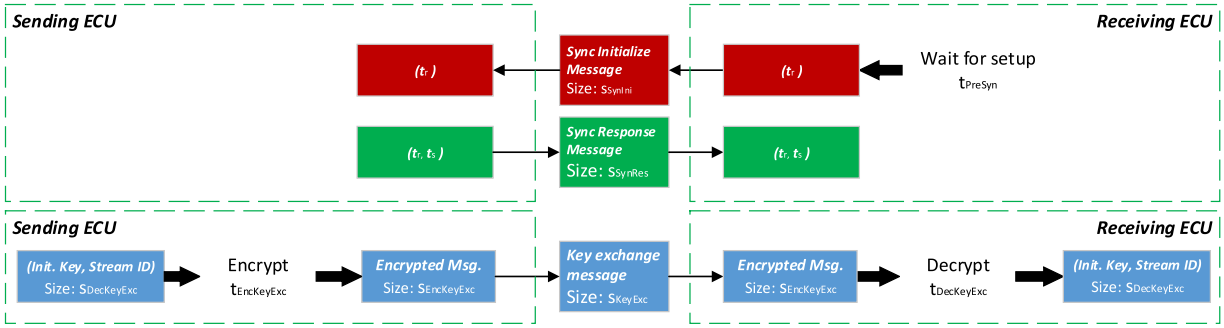

Figure 5.8: TESLA time synchronization and initial key exchange: Timing parameters and sizes parameters that are used in the implementation of this protocol

receiver another timeout t_{EncGra} for the symmetric encryption with the receiver's key is executed. Analogously, for the Deny Message a timeout t_{EncDen} is set. The messages plaintext sizes are s_{DecGra} and s_{DecDen}, while their ciphers are of size s_{EncGra} and s_{EncDen}. The cipher sizes correspond to the sending sizes s_{Gra} and s_{Den}.
Upon reception, a Grant or Deny Message is decrypted symmetrically, with the ECU key, needing the timeout t_{DecGra} or t_{DecDen}.

After authorization is complete, messages that are incoming from the Application Layer are sent by encrypting them with the exchanged session key. Each message, that is sent, has size s_{DecSim} and requires a timeout t_{EncSim} before it is transmitted. Similarly, the received messages need a timeout t_{DecSim}.

5.2.4 Timed Efficient Stream Loss-Tolerant Authentication

All ECUs with the TESLA Communication Module are able to perform communication with the TESLA protocol. Before the data exchange is started, each ECU is preconfigured with all streams that it is allowed to receive and that it is planing to send. Then, the Time synchronization and the initial key exchange are started, followed by the sending of simple messages. The applied sizes and times are shown in Figure 5.8 and Figure 5.9.

Time synchronization: After a time t_{PreSyn}, receivers of a stream initiate time synchronization. This is done by sending a message with its current clock time and the corresponding stream identifier to all senders that are willing to send to it. Senders return a message with the received clock time and their own current clock time. Once the receiver gets a response message, it associates all stream identifiers, that correspond to this sender, with the clock offset value calculated from the received message. Thus, per sender, the clock delay for the according stream is stored on the receiver side.

Sender setup: For each stream identifier a predefined number of MAC keys is generated by the senders. The timing behavior of this step is the following. It is assumed that the time to generate one MAC key is $t_{GenMACKey}$, the number of created keys for stream i is n^i_{MACKey}

and the number of streams for sender k is n^{ik}_{Str}. Consequently, each sender k needs a time $t_{GenKeyTot}$ to generate all keys for all streams. Therefore, at this phase in sender k a timeout of $t_{GenKeyTot} = t_{GenMACKey} \cdot n^{i}_{MACKey} \cdot n^{ik}_{Str}$ is applied.

Initial key exchange: Next, all senders transmit the initial MAC key for each of their streams. To exchange this key, both sender and receiver ECUs use a common key, that is preprogrammed on the ECUs. Alternatively, an asymmetric key exchange can be initiated by encrypting the Key-Exchange message with the corresponding public key. Then, for each stream, the MAC key corresponding to the stream is transmitted to all receivers. Per receiver, the clear key message of size $s_{DecKeyExc}$ is encrypted in the time $t_{EncKeyExc}$ resulting in a cipher of size $s_{EncKeyExc}$, that is only readable to the destination ECU. This ECU is able to decrypt it, needing $t_{DecKeyExc}$. The sending size is s_{KeyExc}. The interval, in which the first key is sent, is considered the first valid interval. Thus, if t^{i}_{Int} is the duration of one interval of stream s_i, from this moment on, till time $t^{i}_{ExpVerMAC} = n^{i}_{MACKey} \cdot t^{i}_{Int}$ messages can be sent and verified.

Sending messages: If the setup is complete and the initial key is sent, the Application Layer is allowed to transmit messages in this stream. Incoming messages of size s_{DecSim} are converted to messages of size s_{Sim}. During this process, a MAC of size s_{MACSim} is generated that is used for the verification of earlier messages. The generation of this MAC takes t_{MACSim}.

Receiving messages: Three conditions are verified when messages are received (see Section 2.3.4). The safe and the new key condition are checked without processing times. Next, a check verifies if the received key with index v is derivable from an earlier key with index k. To verify this, a PRF is applied $n = v - k$ times onto the key, with index k and then, compared to the expected key K_v. This process requires a processing time $t_{VerKeyDer} = t_{PRFKeyDer} \cdot n$, with $t_{PRFKeyDer}$ being the time needed to perform one PRF onto a MAC key. Then, verified messages are stored in a buffer. Subsequently, the messages that are in this buffer are verified with the received key. Thus, per buffer message of size s_{DecBuf}, a validation MAC of size s_{MACBuf} is generated with the received key. This MAC is then compared to the MAC stored in the message. Each of those processes requires a MAC generation time $t_{MACBufVer}$. If there are n_{Buf} messages in the buffer, a total waiting time of $t_{VerBufTot} = n_{Buf} \cdot t_{MACBufVer}$ is needed at this step.

5.2.5 Transport Layer Security

The TLS security protocol is implemented as follows. As it is defined in Section 5.2.2, the whole data exchange passes the Record Layer when sent and received. Therefore, in the following, first the procedures in the Record Layer and then the processes in the Handshake, Change-Cipher-Spec and Application Layer are described. The behaviors of sizes and times of the Record Layer, with and without protection, are illustrated in Figure 5.10, while the one's of the Handshake messages is shown in Figure 5.11 and Figure 5.12.

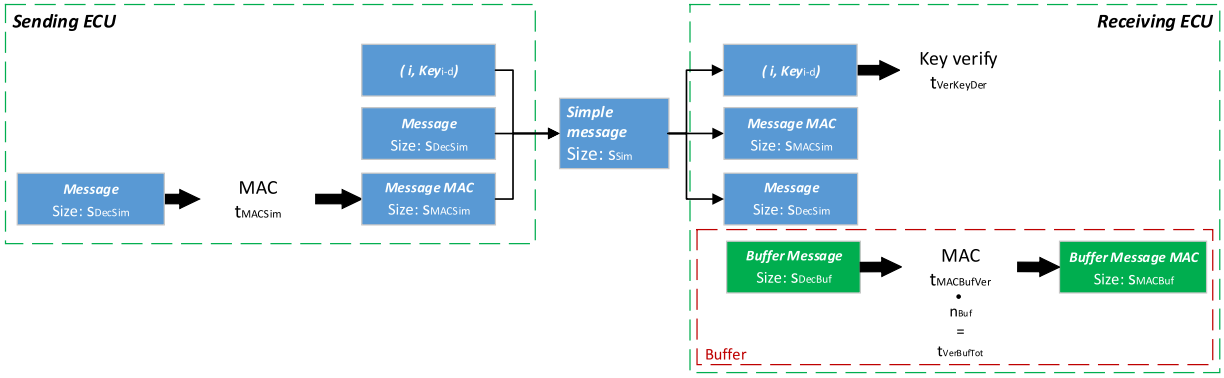

Figure 5.9: TESLA sending and receiving process: Timing parameters and sizes parameters that are used in the implementation of this protocol

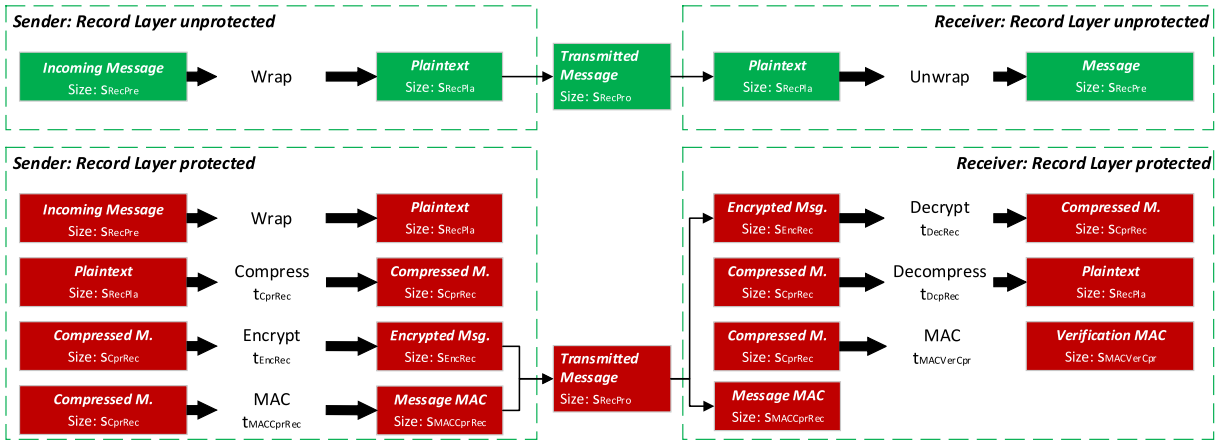

Figure 5.10: Record Layer without and with protection: Timing parameters and sizes parameters that are used in the implementation of this protocol

Record Layer: At this layer no timing delays occur without Record Layer Protection, while with Record Layer Protection compression, encryption and hashing operations lead to delays as they are shown in Figure 5.10. The sent messages change their sizes during this process as it is depicted there. Based on this, higher level protocols are run.

Handshake: During the Handshake protocol, messages are sent and received via the connected Record Layer. At first, there is no cipher-suite negotiated and the Record Layer transmits the messages plainly. This changes once the suite is negotiated. Per stream identifier that is sent, one initial Handshake is to be performed to enable Record Layer Protection for this stream. The sizes and timings that are applied can be read from Figure 5.11 and Figure 5.12. The Handshake involves the following steps.

- *Client-Hello:* During the Client-Hello process the Client-Hello message of size s_{CliHel} is sent to all receivers via the Record Layer. If no session of this stream is available, no timeouts are applied on the Record Layer. Else, previously described timeouts apply. In the following description it is assumed that no suite was negotiated yet for this stream.

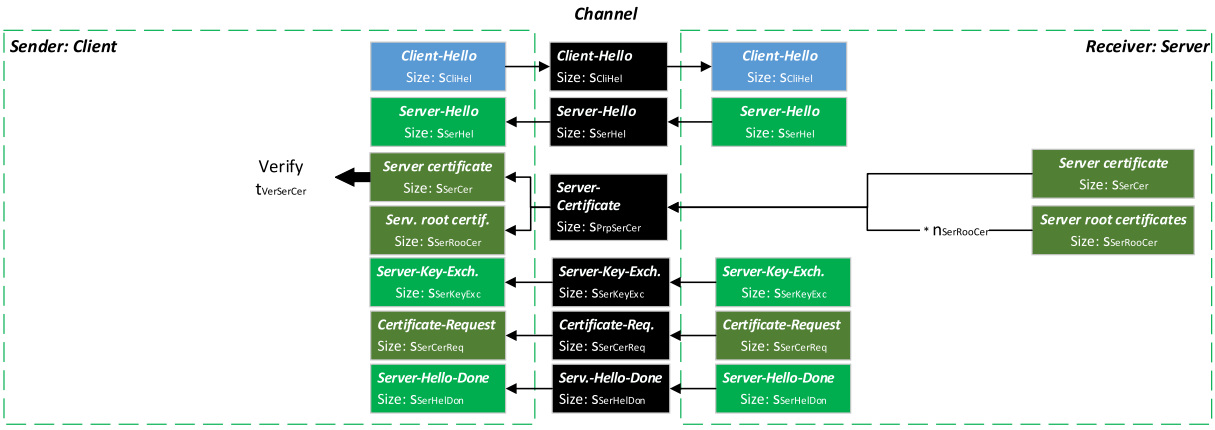

Figure 5.11: Client and Server Hello: Timing parameters and sizes parameters that are used in the implementation of this protocol

- *Server-Hello:* In this step the cipher-suite is determined by sending the messages that are described in Section 5.2.2, with the sizes and delays shown in Figure 5.11. First, the Server-Hello message, the server certificate and all root certificates, each of size $s_{SerRooCer}$, are sent. Thus, if $n_{SerRooCer}$ certificates are sent, the size of the sent Server-Certificate message is $s_{PrpSerCer} = n_{SerRooCer} \cdot s_{SerRooCer} + s_{SerCer}$. After that, further parameters are sent in the Server-Key-Exchange and the Certificate-Request message, followed by the Server-Hello-Done message.
  On the receivers side, those messages are cached without any delays, when they arrive. Once the Server-Hello-Done message is received, however, the incoming certificate is verified in time $t_{VerSerCer}$. After that, the Client-Finished messages are sent.

- *Client-Finished:* According to Figure 5.12, similarly to the server certificate, the client certificate and its $n_{CliRooCer}$ root certificates are sent in the Client-Certificate message. For the Client-Key-Exchange message, the 48 bytes (= $s_{DecPreMas}$) long pre-master-secret is encrypted in $t_{EncPreMas}$ using the public key of the server. Next, the master secret is generated with a PRF. The input of this function is fixed to the length of the random values that are exchanged and by the pre-master-secret length. For the Certificate-Verify message, a plaintext is encrypted with the public key of the server, giving an encrypted message. Subsequently, the Change-Cipher-Spec message is transmitted. Lastly, the Client-Finished message is created with the following content. All cached messages of a total size $s_{DecCliCac}$ are hashed, resulting in a hash of size $s_{HshCliCac}$. The resulting hash is run through a PRF. This results in data of size $s_{VerCliCac}$, that is sent in this message. This message is the first one that uses the Record Layer protection. Thus, the additional timeouts mentioned above apply during this process. On reception of the Client-Certificate message, verification is performed. When the Client-Key-Exchange and the Certificate-Verify message arrive, both times decryption with the private key of the server is performed. Then, the server generates the master secret. After the Client-Finished message is received, a verification hash is created similarly to the previous step.

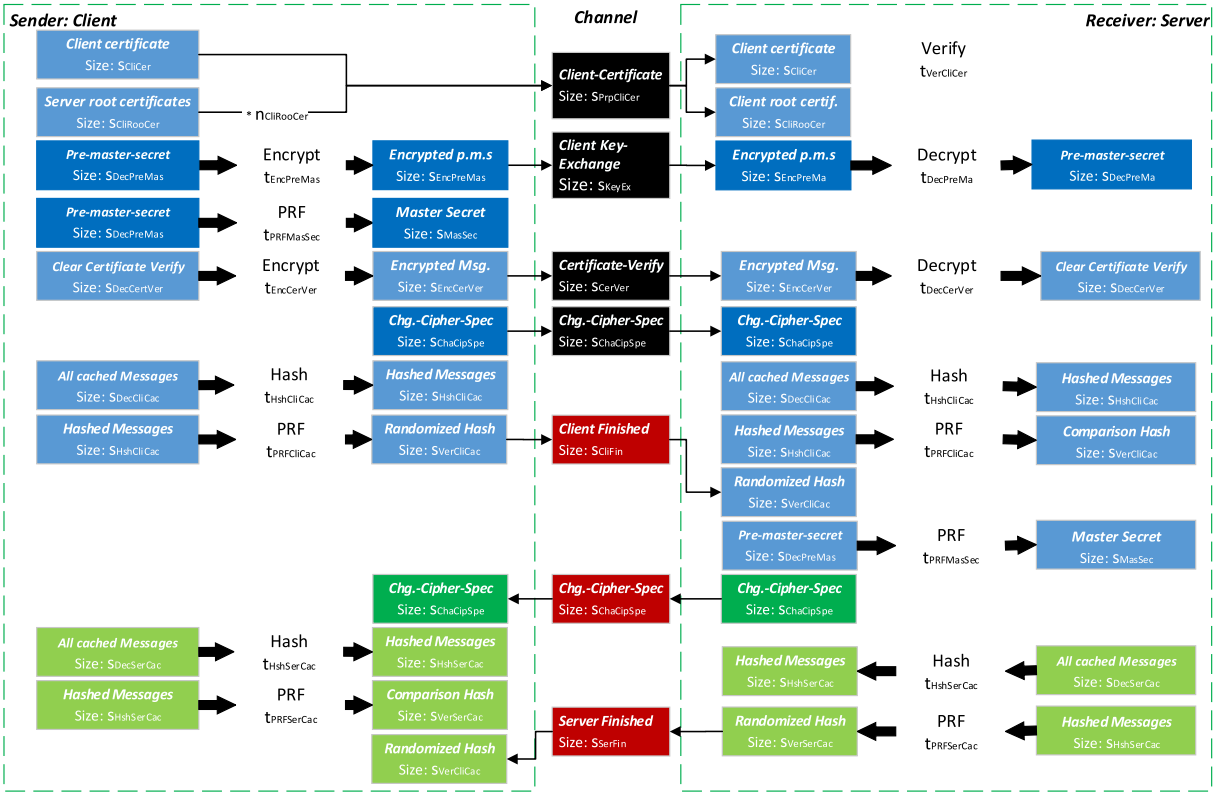

Figure 5.12: Client and Server Finish: Timing parameters and sizes parameters used in the implementation of this protocol. black: no Record Layer protection, red: with protection.

- *Server-Finished:* Then, a Change-Cipher-Spec message is transmitted. This is followed by the Server-Finished message, that creates a hash from all cached messages and uses this hash as a seed for a PRF giving a message. Additional timeouts apply due to Record Layer protection. After the Server-Finished message reaches the client, a verification hash is generated by using all cached messages and applying a PRF with the hash of the cached data as a seed. Then, the data exchange is done via the Record Layer with their corresponding delay times.

5.3 Configuration

As presented in Section 5.2, sizes and timing values of the model decide about the timing behavior of the simulation. Those values depend on each other and on the algorithm parameters with which the model is started. That is why in this section, the configuration of the simulator is presented (see Figure 5.13). First, model parameters are introduced and mechanisms are presented, that are used to set their values. Then, for the three protocols demonstrated in Section 5.2 all parameters are described. Lastly, the relations between those parameters are explained.

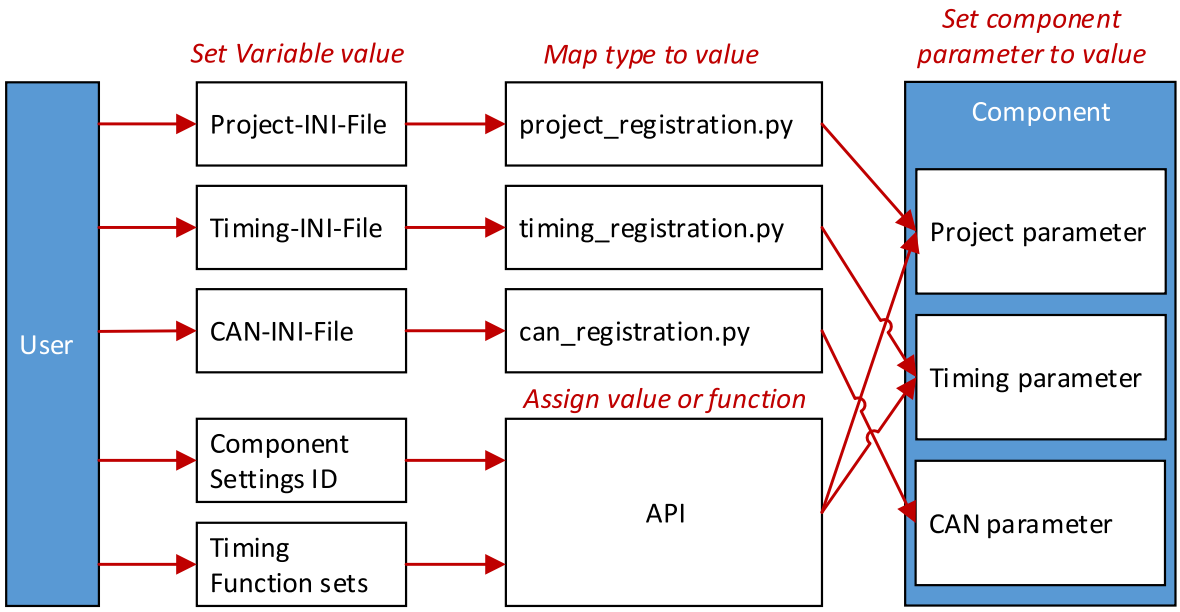

Figure 5.13: State flow to configure project, timing and CAN parameters using INI-files, component settings or Timing Function Sets.

5.3.1 Parameter Types

There are three types of modifiable variables to be distinguished. Those are the following.

Project parameter: Project parameters are used to modify the settings of components and the simulation environment, before the simulation is started. They are set to values or functions. General project parameters are the setting of the message type that is used, the validity of timestamps and nonces, the simulation life time and the data rate of the bus. Further parameters of this type are protocol dependent and are explained in Section 5.3.3.

Timing parameter: Timing parameters define the length of timeouts that are applied in the SimPy processes. Using defined functions, those parameters are mostly determined through the input of project parameters. Thus, those values also depend on the protocol that is used. Nevertheless, they can be set to fixed values by applying the mechanisms introduced in Section 5.3.2.

CAN parameter: Those parameters define the values of CAN message IDs, that are used in the simulation, as they decide about the messages' priorities. Those parameters are important to be able to vary the prioritization of the authentication messages in all protocols.

5.3.2 Configuration Mechanisms

To allow a flexible modification of variables in the simulation the following four configuration mechanisms are realized.

INI configuration: The three mentioned types of parameters are modifiable using INI-files. Each of those types is assigned one INI-file.
Any modifiable variable in the project is registered in the according registration.py file and in the INI-file. The INI-file defines the mode of the variable and the registration.py file maps this mode onto the actual value, that is applied in the code. For example, setting the parameter `alg_1 = RSA` in the INI-file would set the parameter `alg_1` to the mapped value defined in registration.py. This can be e.g. `alg_1 = MyEnum.RSA`. If a constant value is set in the INI file that has no mapping defined, the value in the INI-file is directly applied to the variable. This is desirable, if individual timings need to be adjusted.

Component settings: Values, set in the INI-file, can be overridden using the API. For this purpose, each component in the automotive environment contains a dictionary, that maps a setting ID to a variable in the component. Using this ID, the value is set via the API. For instance, an entry could have a setting ID `alg_1` and an actual variable `self.ecu_sw.comm_mod.alg_1`. Thus, setting `alg_1` in the API changes the variable `self.ecu_sw.comm_mod.alg_1`. Assignable values are either constants or functions that are called on variable access.

Timing Function Sets: Timing variables vary along the input parameters, such as algorithms or data lengths. That is why it is convenient to define an object, that contains the mapping from input parameters to timing values for each timeout of the component. Therefore, Timing Function Sets are introduced. Functions defined in this object are called when the corresponding timing variable is invoked with its input parameters. Their output decides about the length of the timeout to be applied. Using this object, different mappings can be set easily per component.

Protocol Presets: Due to the high number of project parameters in components, presets are used to set those parameters at a higher abstraction level. Thus, for each protocol, a preset is defined. They contain one data structure per group of parameters, that are logically connected. For example, parameters that are used to encrypt a message include the algorithm, the algorithm mode, the key length and the size of the clear text. Thus, by setting those parameters to a value in the preset data structure, the corresponding values in the component settings are set.
Thus, if all four stages are implemented the actual value of any variable is determined as follows. If the setting is only set in the INI-file the value as defined in INI configuration is used. If component settings, Timing Function Sets or presets are applied onto a variable their values are used. If all three are set, the one applied last overrides the former.

5.3.3 Protocol Configuration

For the implemented security protocols various project parameters can be set. Dependant on those parameters, the sizes and the timings in the simulation are determined. Timing Function Sets are used to assign a function to each timing variable, that is defined in Section 5.2. For the calculation of sizes also defined functions are used, which are explained next. Lastly, based on those functions, the parameter relations are described.

Timing function types: Functions that determine the outcome of timing variables depend on various project parameters and sizes. Yet, several timing variables use the same functions, that are described in the following. Those functions determine the value for a timeout by running an SQL command, that looks up the value for the requested input in the database, that is generated from the measurements described in Chapter 4. This generic command depends on the project parameters that are set, the incoming sizes that are used and the cryptographic library (see Chapter 4) that is specified. The following timing functions are applied in the simulation.

- *Bus transmission:* For the calculation of the message transmission time by the bus, the following is assumed. If an unextended CAN frame is sent, the length of it is 44 bits overhead plus up to 64 data bits. The length of a CAN FD frame, however, is defined as follows. It has a 44 bit long arbitration phase part and a 64 byte long data part that is ran with a 8 times higher transmission rate than the arbitration. Thus, it is assumed, that each byte of the data frame needs the same time as one bit needs, if it was sent within an ordinary CAN frame. Consequently, each byte of the data phase is one bit that is sent with the speed of the arbitration phase. Thus, for calculation it can be assumed as an ordinary CAN frame of size 44 bit plus number of data phase bytes n_{data} transformed to bits. On top of that, the transmission time comprises the sending time t_s and the propagation delay t_d. When bit-stuffing is used together with a data rate r in the arbitration phase and an average distance of d between ECUs, the transmission time t_{tot} is

$$t_{tot} = 1,25 \cdot (t_d + t_s) = 1,25 \cdot (\frac{d}{r} + \frac{44 + n_{data}}{r}). \tag{5.1}$$

- *Certificate verification:* To determine the time for a certificate verification several certificate specifications are passed to the timing function that is called once the corresponding timing variable is accessed in the simulation. Those are the hashing mechanism, the encryption algorithm and its mode, the algorithm key length, the number of CAs to the root CA, the certificate size and the signature size. The verification requires three stages. First, a hash is generated from the certificate content. The timing value is requested from the database by passing the hashing algorithm and the certificate size to the database. Next, the digital signature is decrypted and a verify value is requested from the database. For this purpose, the signature size is passed to the database. If RSA is used, also the exponent and the key length are passed, whereas if ECC is used, also the parameter length is passed. The returned value forms the second part for the timeout duration of this process. The last part is determined by hashing the decrypted message. Thus, the last time value needed to verify a certificate is another hashing time similar to the first. Summing up those values gives the verification time for one certificate against one root certificate. Thus if there are multiple intermediate certificates to be verified, this process is repeated. Hence, the time value is the summed value of the three mentioned times multiplied by the number of CAs to the root.

- *Symmetric encryption/decryption:* The time for a symmetric encryption or decryption is determined by requesting a value from the database. The algorithm, the algorithm mode

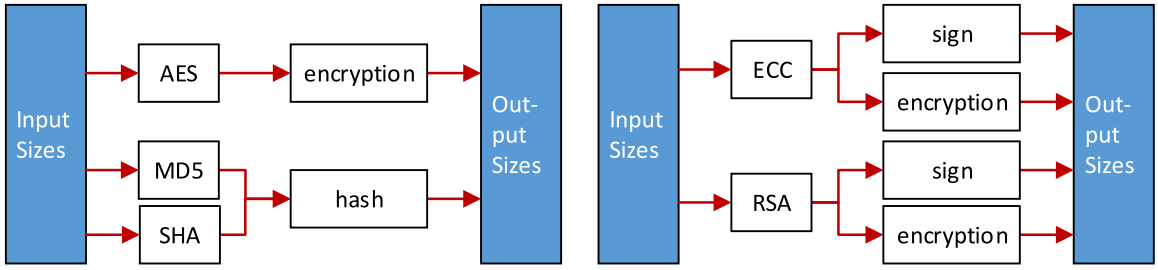

Figure 5.14: Implemented size transformations.

and the key that is used for the operation is passed. For encryption, also the plaintext size is passed to the database, while for decryption the cipher size is input.

- *Public/private en-/decryption:* For RSA the public encryption value is determined by requesting it from the database providing the plaintext size, the algorithm, the key length and the exponent used. If the requested encryption time exceeds the maximum input value of $max = keylength - 11$ bytes for RSA, it is assumed that the incoming message will be chunked. Thus, the encryption time will be multiplied by the number of chunks. Contrary, the timing value in ECC is determined from the database through the plaintext length, the algorithm and the parameter length. Similarly, for private decryption the cipher sizes are input and the decryption value is requested.

- *Signing/verificaton:* Signing is implemented similarly to the public encryption with a request for signing instead of encryption. Also, for verification, this request is the same as the one for public encryption, with the verification tag instead of the encryption tag.

- *Hashing:* To get the time needed to create a hash, the database request is started with the hashing algorithm that is used and the size of the message that is to be hashed.

- *Symmetric key generation:* To generate a symmetric key, the time of a random number generator is measured. The time value corresponding to this process is determined by passing the key length and the algorithm that is used to the database request. The returned value is used for this timing variable.

- *MAC creation:* A MAC creation corresponds to the measurements for AES CMAC that is made with the STM32 Cryptographic library. Thus, the database request includes the data size, together with the key length and the AES CMAC tag.

- *Pseudo random function* A PRF is measured using a random number generator. Thus, the same request parameters are required as in the symmetric key generation case.

Size function types: After a cryptographic operation is performed on a data type in the simulation, the size of the resulting output data will differ from the input data size. Thus, the resulting size depends on the input data and the operation that is applied. In the simulation the following transformations are implemented (see Figure 5.14).

- *Hashing:* The size after a hashing operation is independent of the input length. It only depends on the algorithm that is used. Hence, for MD5 the output data length is 16 bytes, for SHA1 it is 20 bytes and for SHA256 it is 32 bytes.

- *ECC:* The size of an ECC signature is $size = 2 \cdot keylength + 6$ byte. Its encryption output is determined by $size = 48 + \lfloor \frac{inputsize}{16} - 0.01 \rfloor \cdot 16$.

- *RSA:* In RSA input data up to $keylength - 11$ byte is processed and padded to the key length in byte. Thus, when data bigger than $keylength - 11$ byte is to be processed, multiple chunks of the key size are generated as chunking is applied. Hence the output length is $size = keylength \cdot \lceil \frac{inputsize}{keylength-11} \rceil$. The RSA signing operation also leads to the same size as the same operation is performed with a private key.

- *AES:* AES processes data in block sizes of 16 bytes. Thus, incoming data is padded to a multiple of 16 bytes and the output equals the padded size. Hence, the output size is $size = \lfloor inputsize/16 - 0.01 \rfloor \cdot 16 + 16$.

In Section 5.2, all sizes and time values that influence the LASAN, TESLA and TLS protocol in the proposed simulator are described. In this section, so far all possible relations between those sizes, further algorithm parameters and timing values are explained. Based on this, in the following those relations are demonstrated for all protocols in order to demonstrate the generic relations that the simulator provides. This is supported by using tables (see Appendix A), in which the entry Function Type defines which of the above presented database requests is used to acquire the timing value. The field Arguments shows with which parameters this request is started.

LASAN: The relations for this protocol are illustrated in Table A.1 and Table A.2. ECU Authentication starts after the time t_{PreAut} and is repeated after fixed intervals of t_{IntAut}. Project parameters for the SM certificate are the hash algorithm applied $p_{HshSecCer}$, the asymmetric encryption algorithm $p_{EncAlgSecCer}$, its key length $p_{EncKeySecCer}$, its algorithm mode $p_{EncModSecCer}$, the number of CAs $p_{CAsSecCer}$ and the certificate size s_{SecCer}. Analogously, for the ECU certificate, those are the parameters $p_{HshECUCer}$, $p_{EncAlgECUCer}$, $p_{EncKeyECUCer}$, $p_{EncModECUCer}$, $p_{CAsECUCer}$ and s_{ECUCer}. To create the first part of the Registration Message, the asymmetric algorithm $p_{EncAlgReg1}$, its key length $p_{EncKeyReg1}$, its mode $p_{EncModReg1}$ and size $s_{DecReg1}$ are set. The second part of this message applies also asymmetric encryption with parameters $p_{EncAlgReg2}$, $p_{EncKeyReg2}$ and $p_{EncModReg2}$. On top of that, hashing is performed with the mechanism $p_{HshReg1}$. The size of the Confirmation Message before encryption is s_{DecCon}. Also the symmetric encryption algorithm $p_{AlgECUKey}$, the mode $p_{ModECUKey}$ and the key length $p_{KeyECUKey}$ of the ECU key needs to be set. Analogously, for the session key, the parameters $p_{AlgSesKey}$, $p_{ModSesKey}$ and $p_{KeySesKey}$ need to be defined.
The Stream Authorization requires the plaintext sizes s_{DecReq} of the Request Message, s_{DecGra} of the Grant Message and s_{DecDen} of the Deny Message to be specified on program start. Furthermore, if a Request Message is sent, the corresponding ECU waits for an answer to then start

sending. Thus, if no answer is received in a time $t_{ReqWaiRes}$, the waiting is stopped. Lastly, to avoid that too many Request Messages of ECUs are sent, a minimum request interval of $t_{MinReqInt}$ between two requests is defined. During this waiting time, messages that are incoming from the Application Layer can either be buffered or dropped. If they are buffered, all messages in the buffer are sent once a Grant Message is received. The selection of buffering or dropping is set with b_{hold}.

TESLA: The relations for this protocol are illustrated in Table A.3. If a standard TESLA Application Layer is utilized after the time t_{PreSet}, the setup phase is initialized. It is repeated after the defined time t_{RepSet}. During this setup phase the defined number $p_{KeyChaLen}$ of MAC keys is generated once per stream. Next, it is possible to define the PRF $p_{PRFAlgCre}$, the MAC algorithm $p_{MACAlgCre}$ and its key length $p_{MACKeylCre}$ that are used during the creation of the key chain. Analogously, the PRF algorithm used to test the key legitimacy is $p_{PRFAlgLeg}$ and the MAC, that is used to verify buffer messages, is specified via the algorithm $p_{MACAlgLeg}$ with key length $p_{MACKeyLeg}$. The symmetric or asymmetric algorithm, used to send the Key-Exchange message, is defined by the algorithm $p_{AlgKeyExc}$, the algorithm mode $p_{ModKeyExc}$, the key length $p_{KeyKeyExc}$ and the size of the exchange message before encryption $s_{DecKeyExc}$.

TLS: The relations for this protocol are illustrated in Table A.4 and Table A.5. To run the simulation of TLS, it is necessary that the following parameters are set. The algorithms, that are applied once the Record Layer protection is active, are defined by the compression method $p_{CprRecAlg}$, the symmetric encryption algorithm for the reading state $p_{AlgRecRea}$, its key length $p_{KeyRecRea}$ and its mode $p_{ModRecRea}$. In the same way the parameters for the write state are $p_{AlgRecWri}$, $p_{KeyRecWri}$ and $p_{ModRecWri}$. The MAC algorithm is specified over $p_{AlgRecMAC}$ and its key length $p_{KeyRecMAC}$. In a similar way as in LASAN, also the server certificate configuration is defined using the parameters $p_{HshSerCer}$, $p_{EncAlgSerCer}$, $p_{EncKeySerCer}$, $p_{EncModSerCer}$, $p_{CAsSerCer}$ and s_{SerCer}. Analogously, for the client certificate those are $p_{HshCliCer}$, $p_{EncAlgCliCer}$, $p_{EncKeyCliCer}$, $p_{EncModCliCer}$, $p_{CAsCliCer}$ and s_{CliCer}. The size of a root certificate is s_{RooCer}. Next to this, the plaintext sizes for the sent message are to be specified. Those are s_{CliHel}, s_{SerHel}, $s_{SerCerReq}$, $s_{SerHelDon}$, s_{KeyExc}, s_{CerVer}, s_{CliFin} and s_{SerFin}. All other sizes are derived. The PRF, that is used to set the master secret is set via $p_{PRFMasSec}$. Lastly, the hashing algorithm p_{HshCac} and the PRF p_{PRFCac} used for hashing needs to be specified.

5.4 Application Programming Interface

With the model and its configuration possibilities as a basis it is possible to create, modify and start an user-defined simulation of an automotive network by using the API. Those features are described in this section. The setup of the simulation is performed by the API in the following blocks, that are also pictured in Figure 5.15.

Simulation initialization: To setup a simulation project, an AutomotiveEnvironmentSpec object is created, that holds the configuration of the simulation that is to be started. This parameter

receives the maximum simulation time after which the simulation is aborted. At the same time, it is possible to specify if logging is enabled. Also, registration of objects outside of the current folder, that shall be available in the network, are registered to the API. Furthermore, the Monitor object and the result reader are created and connected. They are both needed to extract data from the simulation and are presented in Section 5.5. Optionally, in this step a GUI can be integrated as it is described in Section 5.6. Also, if desired, a stop button can be displayed that allows to abort the simulation at any time. In order to set the configuration of the security protocol, all parameter values that are described in Section 5.3 can be adjusted by generating a Preset object and by applying it to each component Spec.

Component creation: After that, all components of the network are configured and created. This is done by creating a Spec object, that holds the information of the constructor for the component type and a list of identifiers for the components that are generated. This Spec object is different for each created class. Next, this Spec is modified by applying the timing and project parameters that are described in Section 5.2 and Section 5.3. The following Specs are currently implemented.

- *ECUs:* Implementations of AbstractECUSpec are passed the size of the transmission and the receiving buffer and a list of identifiers, that are used for the generated ECUs. On top of that, a jitter value can be applied. This value is multiplied with a random number on all timing parameters of all created component to enable realistic variations in the ECU behavior. Also, a start-up delay for each ECU can be defined. Hence, an ECU is excluded from the simulation until this delay has passed. Moreover, functions can be defined for individual parameters and Timing Function Sets (see Section 5.3) can be added to the ECUs. If the instantiated ECUs use a subclass of RegularApplicationLayer as their Application Layer, sending actions can be defined. Each sending action specifies when a certain message is sent, in which interval it is sent, what message ID is used, what the content of the message is and what the size of it is. Those messages are then sent until the simulation stops. If an ECU with the LASAN Communication Module is applied, it is possible to skip the ECU Authentication phase by setting the ECUs authenticated via the Spec object.

- *Buses:* A Bus Spec contains a list of identifiers for the buses that are created. Based on this Spec they are instantiated. The connection to the ECUs is established using those identifiers and the API.

- *Gateways:* A gateway for the CAN network is created by specifying the SimpleBusCouplerSpec. As a gateway is an implementation of an AbstractECU, it is possible to specify timing and project parameters for them by applying the component settings mechanism explained in Section 5.3. The main timing parameter, that is specified here, is the delay of the gateway. Thus every message, that is incoming from one bus, is forwarded to all other buses after the specified delay.

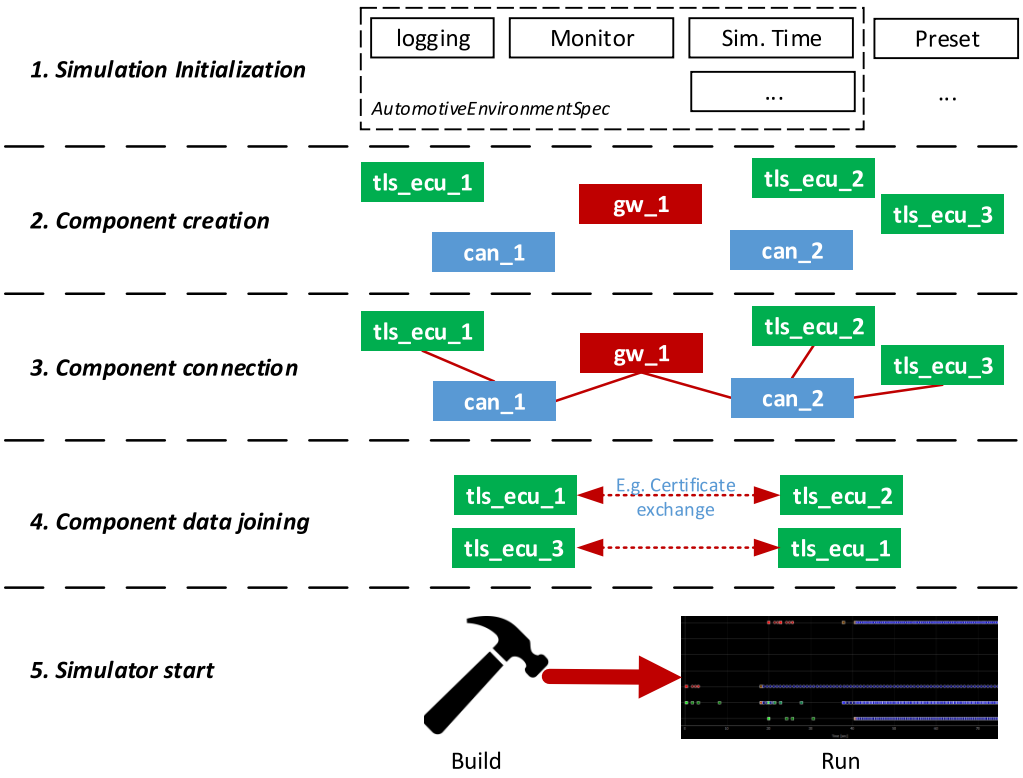

Figure 5.15: Steps to setup a simulation.

The configured Specs are used to add a defined number of components with this Spec configuration to the environment. This is done by passing the number of components to create, the class of the component and the Spec object to the AutomotiveEnvironmentSpec.

Component connection: The configured ECUs and gateways are connected to the created buses. This is done by passing the bus identifier and a list of all ECUs and gateways that should be connected to this bus to the API. Hence, by linking ECUs to specified buses any ECU distribution can be achieved. Also, by passing gateways to multiple buses, any constellation of interconnected buses can be created.

Component data joining: The following step is used to further configure the components, which also has to be performed at design time in actual hardware implementations of intra-vehicular networks.
For some protocols, data of one component needs to be known to another component. For LASAN, it is required that the SM has all intermediate certificates to verify the ECU certificates and vice-versa the ECU has all information to verify the SM certificate. Thus, in this stage the certificates are generated using the Certificate Manager, that is introduced in Section 5.1.3. Then, using the API, the required certificates are passed from one party to another. On top of that, each component needs to get the information about which streams it is allowed to be sent. This information is passed to the respective components in this phase. Lastly, an ingress and an egress filter can be installed in the gateway that filters not allowed message IDs.

Simulation start: After all configurations are finished, the AutomotiveEnvironmentSpec contains the information necessary to create the environment. That is why in this step all information, that is given in this object, is applied to the components. This includes for instance the configuration of component settings by their identifier or the actual connection of the buses. Based on this, the simulation is run as described in Section 5.1.

This section describes the API, that is implemented in order to access and control the model. It consists of five phases that allow to create user-defined architectures and components. This API can also be used to define the input and the output data of the simulator, which are described in the next section.

5.5 Input-Output Processing

In this section, first, the possibilities to input data are summarized and next, the mechanisms that are used to create the output are explained.

5.5.1 Data Input

To configure the simulator defined inputs need to be passed to the simulator. Those are on the one hand parameters that are passed during the project configuration phase and on the other hand values that are input via the access to a database.

Project Configuration: In Section 5.3 and Section 5.4 the main ways to input data are discussed. Using the described mechanisms, user-defined project and timing parameters are applied to modify the system. Moreover, the API is used to build the architecture and to setup the simulation with the desired components.

Database Requests: Next to those input mechanisms, the database is used to define the relations between project and timing variables. The applied SQL database consists of one table with nine columns. It is populated with data, that is measured from the CyaSSL, as well as the STM32 Cryptographic Library in hardware and software mode, as it is described in Chapter 4. That is why the first column specifies which library was measured. Next, in the second column, the type of measurement (i.e. hash, encryption,...) is specified. In the further fields, the algorithm, its mode and its key length are given. The exponent for RSA and the parameter length for ECC are defined. Lastly, there are two columns for the size that was measured and the time it took to perform the operation on it.
This database is integrated into the simulator using a Singleton [Bis07] object, that represents the database file and the access to it. When requesting data from it, the inputs described in Section 5.3 are converted into a SQL statement, that is used to retrieve data from the database table. This database concept offers the chance to easily run the simulator with any set of measurements. Hence, if new measurements are made with a different library than the one proposed,

the according measurements only need to be added to the database and the lookup tag for the library needs to be set in the according Timing Function Sets.

5.5.2 Result Output

To read results from the simulator, at multiple stages of the simulation objects are created that gather state information throughout the runtime of the simulation. Those results are then processed by passing them to a Monitor or by being output directly. In the former case Monitors process results via an Input-Handler-Chain that forwards data further to the GUI or writes it to the file system, while the latter outputs it directly. All components that are involved in those two processes are explained in the following.

Monitor inputs: After the simulation process started, the states of the simulation variables change with every timeout that is performed. In order to track those changes of states, the data structure Monitor input is defined. This data structure carries information of a component of the environment (ECUs, buses, gateways) at one point in the simulation time. It contains Monitor tags that define the type of event that occurred and at which stage of the communication process it happened. For instance one tag is defined for a certain ECU at the start of the encryption of a message and one tag after it is done. This enables the user to track the plot of the simulation after it is finished. Next to Monitor tags, a Monitor input can store the invoking component identifier, the identifier of an associated component, a message ID, a message, a message size and a stream identifier.

Monitor: Processing the gathered Monitor inputs is implemented with the observer pattern [Bis07]. For this purpose, an observing Monitor object is created. This object coexists with the other components and does not interfere with them. Each component, that wishes to deliver Monitor inputs to this object registers to it. On top of that, all objects that want to receive the gathered data can subscribe a method to defined handlers inside of the Monitor. Next, in parallel to all component processes, two observer processes are started. The first process requests and stores the gathered list of Monitor inputs from all registered components in defined simulation time intervals. In contrast to that, the second process regularly passes this collected data to all its handlers, that again forward them to their subscribers.

Input-Handler-Chain: An individual handling of Monitor tags is enabled by the handler chain, that is implemented using the Chain-Of-Responsibility pattern [Bis07]. In this pattern, data, that is incoming from the Monitor on every publishing, is passed to one handler object that passes it to further handlers along a chain. Each handler object defines the Monitor tags, that are relevant to it. Depending on those tags the corresponding handler extracts the Monitor inputs that are relevant to it and publishes them to all connected subscribers.

Interpreter: Interpreter objects, as it is depicted in Figure 5.16, subscribe to handlers and thus, regularly receive the filtered Monitor inputs from them. Given this data, those either push

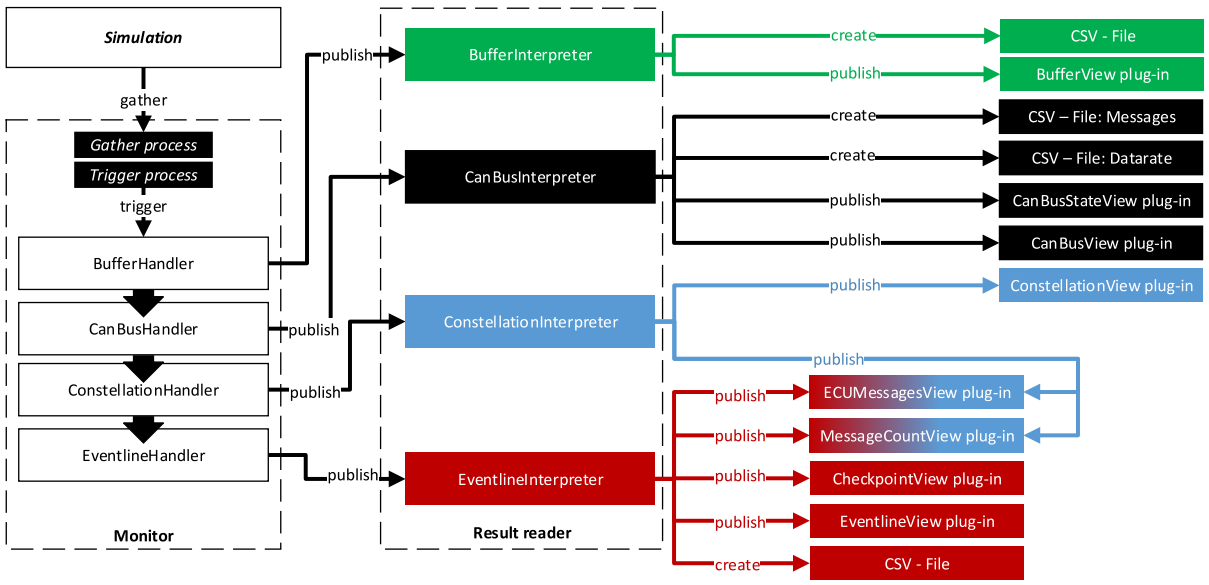

Figure 5.16: Observer Monitor retrieves results from all components and pushes results to the Interpreters. Those forward them to files and GUI plug-ins.

the data further to connected GUI plug-ins or write their results into a file. They are again extensible as they are also implemented using the Factory pattern [Bis07]. Thus, in order to add a new Interpreter, a class needs to be created, that inherits from AbstractInterpreter.
The management of those Interpreters is done via the Result Reader object. Thus, upon simulation generation, first a Result Reader object is created. Then, in this object, all Interpreters that are to be used are enabled. At the same time, a list of options and a save path is provided for each Interpreter. If a GUI plug-in is enabled the corresponding Interpreters are automatically activated. The connection to the simulation is established by connecting the Monitor object to the Result Reader using the API. Currently the following Interpreters are implemented.

- *Buffer Interpreter:* This object receives data from the Buffer Handler. It contains the states of the receive and transmit buffers of each ECU connected to the Monitor. This Interpreter writes the state of the current buffer utilization in kilo byte to a Comma-Separated Value (CSV)-file, on every publish call of the Monitor process.

- *Can Bus Interpreter:* This Interpreter picks up data directly at each CAN bus. Therefore, every frame that is sent on a bus generates two Monitor inputs. One before it is sent and one after it is sent. Those tags are written to a CSV-file together with all information given in the Monitor inputs. On top of that, the saved tags make it possible to calculate the data rate. This is done as follows. New Monitor inputs are incoming in the regular publishing interval t_{pub}. On top of that, each Monitor tag contains the size s_{frame} of the frame that is sent. If in the interval t_{pub}, n_{frames} frames were pushed to the Interpreter, the transmission rate r_{bus} for each bus can be approximated with

$$r_{bus} = \frac{n_{frames} \cdot s_{frame}}{t_{pub}} \tag{5.2}$$

Consequently, in addition to the Monitor inputs also this calculated data rate is outputted to a CSV-file. On top of that, if specified, both tags are pushed to the GUI.

- *Constellation Interpreter:* This Interpreter receives the information about the system architecture from the Constellation Handler exactly once on the start of the simulation. Then, it pushes this information about all buses, ECUs and gateways, as well as their connections and settings to a GUI plug-in.

- *Eventline Interpreter:* This object receives its data from the Eventline Handler. It is responsible to track all events that occur within the Communication Modules for LASAN, TESLA and TLS. For this purpose, all events, as they are described in Section 5.2, are stored together with their Monitor tags. Based on those tags, all events are written to a CSV-file together with their time of occurrence and all parameters of the Monitor input. In LASAN, for instance, one entry is written each time an ECU sends a Confirmation Message and another tag is output when the confirmation is encrypted. Outputs of this Interpreter allow to exactly comprehend all processes, that took place in the simulation.

Direct Output: The problem with applying the Monitor principle is its complexity, which results in worse simulator performance. This is because in the gathering process an iteration over all connected components is performed and repeated periodically. Also, during the regularly executed publishing, multiple iterations over all Monitor inputs are performed in order to filter them at the handlers. This is why optionally direct outputting can be enabled. If this option is active, no Monitor is present. Instead all Monitor inputs are written directly to a CSV-file right in the moment they occur. As a feature, a list of Monitor tags can be passed to this mechanism, that defines which Monitor inputs are to be stored. This implementation results in a better performance, but requires to perform further interpretations based on the received CSV-file.

5.6 Graphical User Interface

To be able to better understand the outcomes of the simulation and the processes that took place in it, it is necessary to visualize the data of the simulation. For this reason a modular GUI is implemented. In this section first the GUI components are explained. Next, their execution and integration with the API is described, before then, all implemented GUI plug-ins are presented. Lastly, as a conclusion to this chapter, all extensible components of the simulator are presented.

5.6.1 Interface Composition

The interface is composed of components that shape the main window and plug-ins that can be used in order to create user-defined views of the simulation. They are explained in the following.

Components: The GUI is realized using Qt 4 and comprises a section that describes the simulator, a combobox to activate plug-ins, a text box to configure the arrangement of widgets,

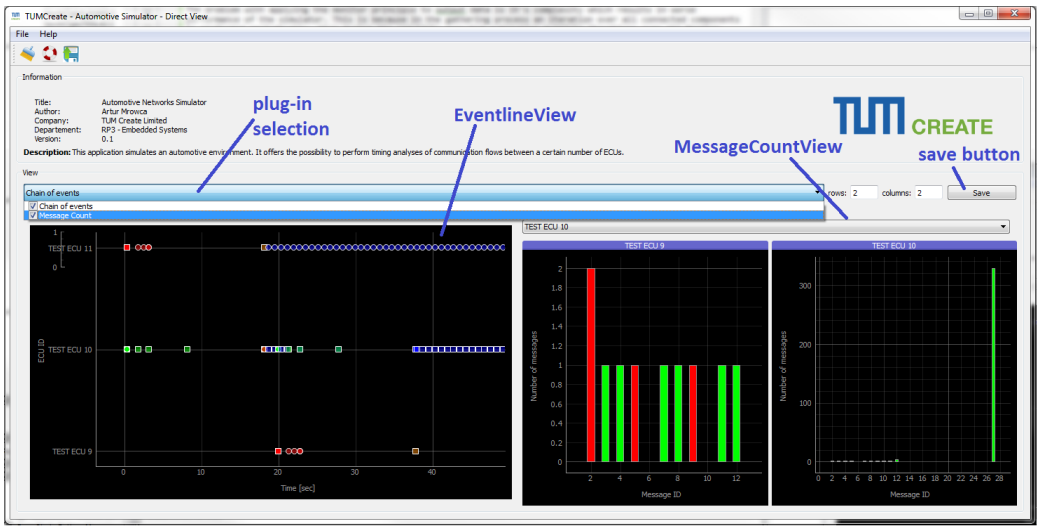

Figure 5.17: Screenshot of the GUI main window.

the widgets provided by the plug-ins and a button to save the results of all widgets. The main window is illustrated in Figure 5.17.

Plug-Ins: Each view is implemented as a plug-in. In terms of code this means that each view is defined as a class that inherits from AbstracViewerPlugin. Thus, all plug-ins located in a certain folder can be created. On creation of the simulation, it can be specified which of the available plug-ins should be used during the simulation, leading to those plug-ins being available within the main window. In order to function with this window, each plug-in provides methods to communicate with the GUI. First, this is the name of the combobox that is specified. Next, this is the QWidget object, that is seen on the main window when this plug-in is activated in the combobox. Furthermore, a list of Interpreter classes is provided, that push their Monitor inputs to the view. If any of those Interpreters publishes, this plug-in's update method is called and the Interpreter Monitor inputs are passed to the view. Based on the received data, this method then updates the view of this plug-in. Lastly, a save and a load function are implemented, to be able to save the widget data to a file and to load it to the view again.

5.6.2 Functioning

After the components were described, next, the integration of the GUI with the whole simulation system is explained. This comprises the execution of it and the API connection which are presented below.

Execution: In order to not interfere with the running simulation, the GUI is started in a separate thread. Consequently, the communication to the simulation is established solely by connecting each GUI plug-in to the respective Interpreter, that is specified in the plug-in. Hence, each time the corresponding Interpreter publishes, its data is pushed to the plug-in's update method, that updates the corresponding widget. As the GUI is running in a different thread as

the simulation, the Signal-Slot mechanism, offered by Qt, is applied to provide thread safety [The15]. As those updates happen in regular simulation time steps, a live view on the events in the simulation is possible. On top of that, this design allows to choose the widgets that are to be displayed during run time. To save the visual results of the simulation, the save button can be used to write the results to a file. Consequently, using the corresponding method, results can be opened from that file in another GUI window at any time.

Integration with API: The integration of the GUI into the simulation is non-intrusive. Two steps are performed to integrate the GUI to the simulation. First, the GUI window is generated. Next, the Result Reader object that holds all Interpreters is passed to the window, together with a list of GUI plug-ins, that are to be displayed. Consequently, the GUI window generates all plug-ins and connects each of them to the corresponding Interpreter by subscribing to its publishing method. The according Interpreters are enabled automatically in the Result Reader to guarantee that the plug-in is fed with the requested data.

5.6.3 Realized Plug-Ins

The following plug-ins are implemented. In Figure 5.16 the Interpreters, used to provide information to those plug-ins, are pictured. Their screenshots are also shown individually in Appendix B. The *Buffer View* displays the state of the transmit and receive buffer. Next, the *Can-Bus-State View* shows all CAN buses on the axis of abscissas and the time on the axis of ordinates. Then, for each message that is sent on the bus one dot marks a frame, that is sent on this bus. Moreover the *Can-Bus View* illustrates the data rate of each CAN bus in a time-data-rate diagram. In the *Constellation View*, all components of the networks and their connections are shown. By clicking on any component its settings are displayed. Furthermore, the *ECU-Message View* shows all information of all events that occur in one ECU. By selecting the corresponding ECU, in and outgoing messages, as well as ongoing processes can be investigated for one ECU. By using the *Message-Count View* the number of sent and received messages can be illustrated for each ECU. That is why, for one selected ECU a histogram shows the message ID on the abscissa and the number of messages on the ordinate. Different colors for the columns are used for sent and received messages. In the *Checkpoint View* all Monitor inputs, that are outputted from the Eventline Interpreter, are written into a table. This table is able to filter values. By selecting an ECU and an associated ECU, the table only shows Monitor tags that refer to the communication between those two components. Lastly, the *Eventline View* shows all Monitor inputs returned by the Eventline Interpreter in a diagram with events. This diagram has the time on the x and the ECU identifier on the y-axis. Thus, every event that occurs in one ECU is depicted on the x value that corresponds to this ECU. By clicking on the events, further information, such as the type of message, its size or its content are displayed.

So far the GUI with all its components, interaction and plug-ins is discussed in this section. Next, the extensible implementation of this simulator is emphasized by naming all extensible parts of the simulation in the following section.

5.6.4 Simulator Extensibility

A major advantage of the implemented simulator is its extensibility. That is why in the following, the interfaces for extensions are presented.

ECU/Bus implementations: The generation of ECUs or buses in the simulation is based on the Factory pattern [Bis07]. Thus, every component implementation can be created using an identifier for the ECU or the bus. On program setup all implemented components, that are in a specified folder, are determined. Then, by providing the class name to the API those components are integrated in the simulation in the way described above.

Layer implementations: In addition to the components themselves, it is also possible to reimplement the individual layers of an ECU. For each layer, it is again only required to inherit from the given abstract class of the layer. Then, this layer can be set in any ECU, that is created or in the Communication Module that is used.

Configurations: By applying the configuration options that are described in Section 5.3, all components and layers are able to apply modifiable and extensible timing and project parameters. In particular the timing values and their parameter relations can be extended by adding entries to the database that connects them (see Section 5.5).

Result outputs: To output results, incoming data is filtered according to a handler chain. This chain can be extended by implementing a handler and by adding it to the chain. One step further, the filtered data is passed to Interpreters that are also implemented as plug-ins. Thus, the generation of a new Interpreter specifies which handlers it wants to use and gets the data filtered by them. Those Interpreters are also usable, if they are in the specified folder and if they inherit from AbstractInterpreter. This is detailed in Section 5.5.

GUI plug-ins: The GUI can be extended by simply providing an implementation of AbstractViewerPlugin in the right folder. Next, by specifying the identifier for the created plug-in during the simulation setup, this view is activated in the GUI. Details of this process are described in Section 5.5 and Section 5.6.

In this chapter a discrete-event simulator is proposed that allows to validate protocols and architectures on a conceptual and temporal basis. This is done by first describing its main components and their configuration. Following this, the API is discussed that is used to set up those components. Then, ways to configure and to retrieve results from the simulation are presented. Lastly, the GUI is presented that can be used to visualize the simulations. This simulator is used to test various architectures and to optimize the simulator's performance. Also, it is validated using software tests. This is detailed in the next chapter.

6 Testing and Optimization

Intra-vehicular networks can have different architectures. Thus, in order to generate valid statements about the quality of the simulation and its applications, an evaluation framework is required that takes into account changes of the communication system architecture. That is why in Section 6.1 a test case generator is introduced. This generator is implemented to generate results for systems with various architectures. This evaluation framework is then utilized to generate test cases that are used to verify the implementation of the simulator, which is described in Section 6.2. Based on the outcomes of those test cases, bugs can be detected. On top of that, this generator is used to compare different versions of the simulator in order to optimize its computational performance. This is detailed in Section 6.3.

6.1 Test Case Generator

The test case generator (see Figure 6.1) is implemented on an existing basis and is extended in the course of this thesis. The part that was existing in the generator comprises the generation of test cases for the LASAN protocol under various architectures by reading in information from the command line. The architectures are generated according to random distributions. This is extended by adding the implementations for TLS and TESLA to the environment in a similar way as it is defined for LASAN. This includes the creation of the according layer and ECU implementations. On top of that, for all three protocols phases were added to the system that execute the steps of those protocols sequentially.
This test case generator is used to generate variable test cases from the command line. Thus, by starting this script with different arguments, various architectures are created. In the following, first, the inputs for the generator are described. Next, the generator's execution is explained, before lastly, the test case generation of each implemented protocol is presented.

6.1.1 Generator Inputs

The generator is based on settings, that are defined in the Architecture Configuration object, settings that are passed by the user over the command line and a random number generator that ensures the diversity of test cases.
The presets, as they are defined in Section 5.3, specify the protocol settings that are used in the simulation. On top of that, all GUI plug-ins that are to be started and all Interpreter options can be set. If direct output is used, all Monitor tags, that are to be logged, can be set as well.
Further simulation specifications are passed via the command line. They provide component settings including the size of their buffers, the measurement library that is applied for their timings and an option for LASAN to define if the ECU Authentication phase is to be skipped. Next to this, also project settings are specified. They comprise the maximum simulation time, the protocol that is to be used in the Communication Module, the enabling of logging, the displaying of the GUI and the output paths for the results. Moreover, additional arguments are passed to configure the random number generator. Those include the minimum and maximum number of ECUs, buses and messages. Also it provides further parameters comprising a random seed, random factors for the number of receivers per stream and random factors for the receivers per ECU. Lastly, the command line input defines if the simulation is started in Standard or Rapid mode. The Standard mode includes the Standard bus and the application of the Monitor design. In contrast to that, the Rapid mode uses some simplifications and a direct output. Those simplifications are described in more detail in Section 6.3.

6.1.2 Simulation Generation

Given the described inputs, the simulation is created via the API as follows. In the first step, the random number generator is used to define the number of ECUs n_{ECU} and buses in the network, depending on the range that is passed via the command line. Next, the ECUs for the corresponding protocols are instantiated. The type of Data Link Layer that is created depends on whether the Rapid or the Standard mode is selected. Also the Application Layer and the Communication Module corresponding to the selected security protocol are instantiated. Furthermore, the ECU is configured using the defined settings. Subsequently, the specified number of Rapid or Standard buses is created. If multiple buses are used they are interconnected via one central gateway.
Those configured ECUs are then assigned randomly to one of the instantiated buses. After that, the number of message streams that has to be generated is determined via the random number generator by inputting the receivers per stream factor r_{rps} and the randomization factor r_{ran}. Then, for each of those streams i the number of receivers $n_{rec,i}$ is defined using

$$n_{rec,i} = r_{rps} \cdot (n_{ECU} - 1) + random(-r_{ran}, r_{ran}). \tag{6.1}$$

Furthermore, a random sender ECU and the found number $n_{rec,i}$ of receiver ECUs are assigned to each stream. For this purpose, a random number generator with a Medium Average Distribution (MAD) is used. The sizes and time intervals of those sent streams are also chosen randomly. In this case, distributions that are defined by a lower and upper bound are given.

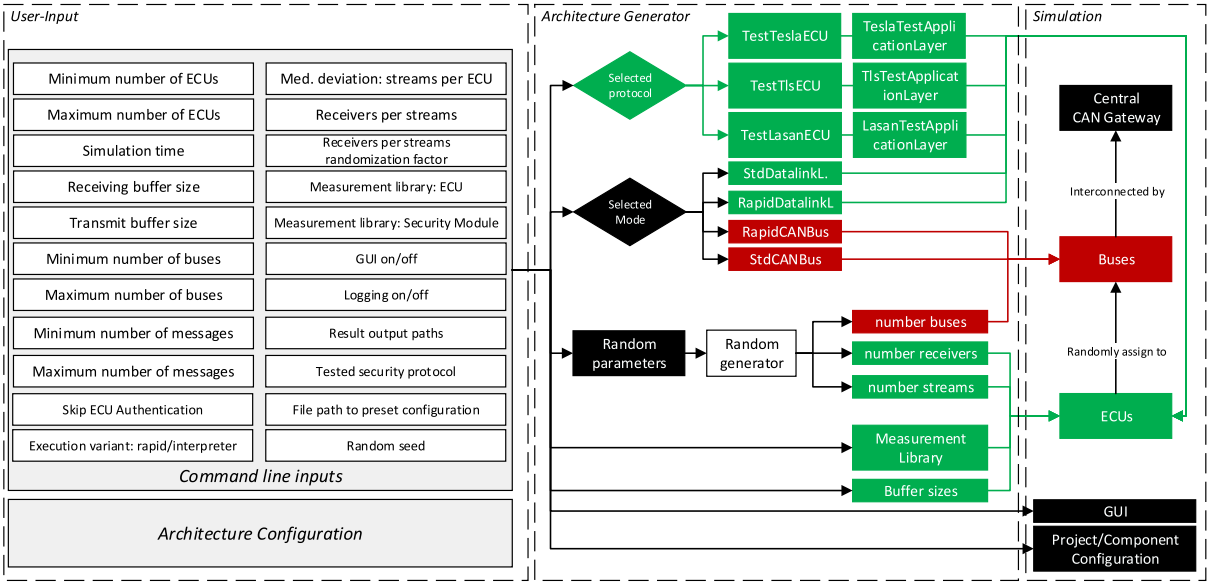

Figure 6.1: Structure of the test case generator.

Consequently, a random value determines which distribution is to be used. Then, the size and interval value is determined by running a random number generator within the boundaries defined by the assigned distribution.

6.1.3 Protocol Processes

So far a general introduction to the test case generator was given. Next, the processes of the simulation are described, when the test case generator is started with different protocols.
Each protocol applies a AbstractECU implementation including an AbstractApplicationLayer, that is created for this test case generator. Those define the processes in each ECU and are described below for the three implemented protocols.

LASAN: The implementation of LASAN for the test case generator works as follows. For each stream identifier that this ECU is sending, its Application Layer starts an own sending process. This process includes two phases. On start of the simulation a Request Message is sent for this stream. This happens either after the ECU Authentication is complete or, if it is disabled, immediately after the simulation starts. This Request Message is repeated, if no response is received within a certain time. After the request is answered, the sending of messages is started. For this sending process, the following abstraction is applied. The traffic of stream messages is only less affected by normal messages, as stream message IDs are higher prioritized than normal messages. That is why, in order to improve the simulator performance, for large scale systems only a defined number of stream messages is sent. Furthermore, by setting an additional option the Request Message can also be started after a random time passed.

TESLA: This protocol is implemented similarly to LASAN. One process is started per stream at the Application Layer of the respective sender. Then, according to the protocol definition,

for each component the timing synchronization is started. After this phase is complete, the Application Layer initializes the setup phase including the key exchange. Consequently, each sender waits per stream until all receivers acquired the key. Once this is is the case for a stream, the sending process corresponding to this message ID starts to send its messages. Analogously to LASAN, the number of messages is also limited.

TLS: This protocol is implemented in a similar manner to the other two protocols. In the TLS implementation at the beginning of the simulation the Handshake is initiated. Followed by that, the Application Layer waits for this Handshake to finish. Conclusively, it sends a defined limited number of messages.

In this section the test case generator is introduced, that enables to create various random simulation scenarios depending on specifications about the authentication protocol and the architecture. This generator is used to test the software components of the implemented protocols. That is why, in the next section, the testing of those software components is described.

6.2 Software Test

The procedure that was used to develop the software system follows the waterfall model [Som11]. In this iterative approach, first, the requirements of the software are analyzed. Next, an initial design is created, followed by its implementation. Once this implementation is complete, it has to be verified by testing the software thoroughly. Based on the outcomes of those tests, the performance, as well as the stability of the program is improved and errors are fixed. The tested components are then integrated into the overall system and the iteration is started anew with the requirements for the improvements. For this purpose, software tests are realized. Those are performed, in order to validate the correct functioning of the simulator and are described in this section.
First, the test environment that is used for validation is presented. This environment is then used to perform tests, grouped in four test sets. These are explained next. Lastly, manual tests that were performed are described.

6.2.1 Framework

The framework that is used for testing is the Python unittest module. This environment allows to test multiple test cases together by grouping them into test sets. Typically a test case consists of four steps. First, in a preparation phase the system is setup to its initial test scenario. Next, the expected outcome for the performed test is determined. Following this, the system is run and the parameter under test is extracted giving the actual value. This actual outcome is then compared to the expected result. If they are equal, the test case is successful, else it failed. This principle is applied in the four implemented test sets, that are depicted in Figure 6.2 and described in Section 6.2.2.

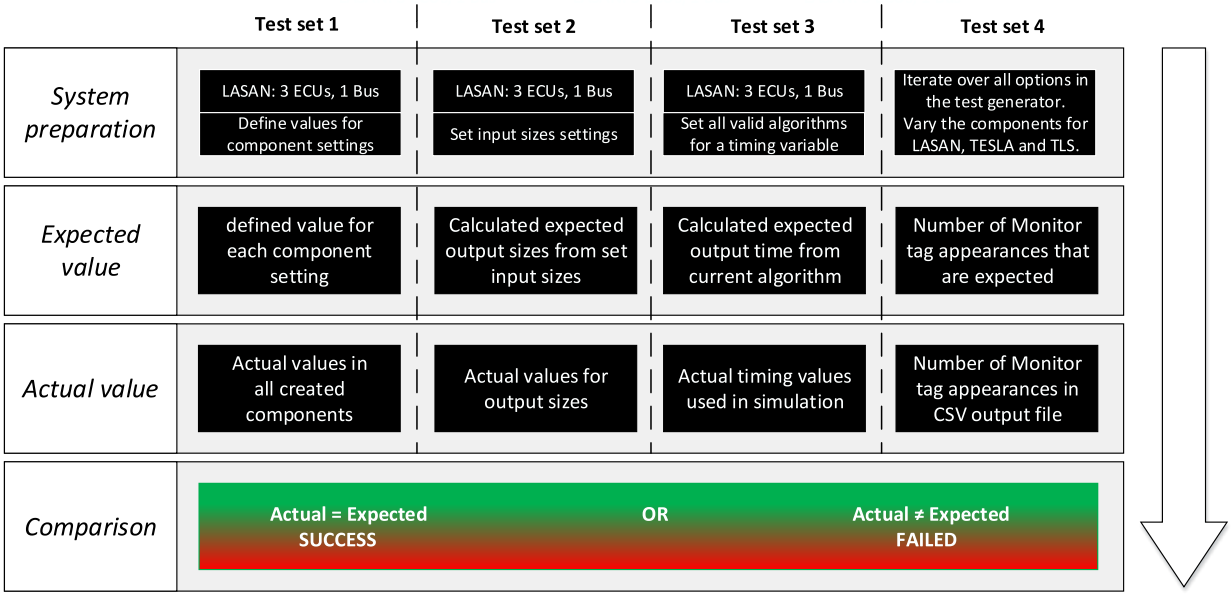

Figure 6.2: Schematic of implemented test sets.

6.2.2 Test Sets

The following test sets are implemented and used to validate the software. The four test sets that were successfully tested are explained hereafter and illustrated in Figure 6.2.

Test set 1 - Testing component settings: In order to verify that all component settings are applied in the simulation when the API sets them to a value, this test set is used. For this reason, first, a LASAN environment is created via the API. The test environment comprises three ECUs on one bus as this is sufficient to check the component settings. This is because, solely the application of those parameters is checked. It is also sufficient to perform this test only for LASAN, as the tested principle works analogously for other protocols and components as well. During the configuration phase all available component settings in all ECU objects and the SM are set to a random value. Each of those expected random values is stored. Next, the simulation is built. To now validate if the settings are applied, the stored expected values are compared to the actual values of the ECUs in the built simulation. If all component values are equal, the test is successful. In any other case the test fails. This test set consists of one test case for both the ECU and the SM implementation.

Test set 2 - Testing size parameters: During runtime, the sizes of messages in the implementation are determined in two ways. Either they are set before the simulation, using the component settings or they are calculated from other fixed sizes. That is why in this test set all sizes that are derived from other sizes are verified. Consequently, the system is setup similarly to the first test set and all input sizes are set using the component settings. Next, the system is started and the actual sizes that are applied in the code are logged. This logged data is then compared to the expected sizes of the individual size parameters. If the size values correspond

to the expected value, the test is successful, else it fails. Here, each size parameter that occurs in the simulation corresponds to one test case.

Test set 3 - Testing timing parameters: In this test set all timing variables are validated. This is again done by setting the project parameters, that are the input for the timing variable under test. Next, the expected outcome is calculated from the applied parameters, which are input sizes, algorithms and algorithm parameters. Subsequently, again the simulation is started. The actual value of the timing variable under test is logged and the outcome is compared to the expected value. Depending on the outcome of the comparison of those values, the test case is successful or fails. Each test case represents one timing variable in the simulation and sets all parameter configurations that are possible for its function input.

Test set 4 - Protocol outcome tests: To be able to check if the simulation behaves as expected this group of tests is created. By applying the test case generator that is presented in Section 6.1, in Rapid mode different project values can be set and various architectures can be created. Depending on the input parameters for the generator, certain numbers of Monitor tags have to be present in the CSV-file that is resulting from the test. That is why this test set applies the test case generator multiple times, with different settings and checks if the resulting CSV-file contains the expected values. Hence, an iteration over all parameters that are variable in the simulation is performed. First, for each iteration the command line arguments are determined. Next, the test generator is started via Python from the command line, with those specified settings. Then, the generated CSV-file output is checked by counting the occurrences of each Monitor input and comparing it to the expected number of outputs. If all expected Monitor inputs are present in the right number, the iteration is successful and the simulation is tested with the next set of command line arguments. Those arguments include all available libraries, encryption methods and hashing algorithms, as well as their parameters. Furthermore, they comprise defined numbers for the ECUs, buses and messages and various random factors. If all iterations with all architectures deliver the expected results, the test case is successful and fails otherwise. This principle is implemented for each protocol as one test case.

6.2.3 Manual Tests

Lastly, also manual testing was performed. This is done through the implementation of one test case for each protocol. A minimal system of up to five ECUs on two buses connected by a gateway is investigated, where up to 5 messages are sent between the units. On any integration of new components, this system is used to check if the right behavior is achieved. This investigation is further enhanced by the inspection of the resulting output that is provided by the GUI. If discrepancies are discovered in the plots, the system is optimized and the tests are repeated.

6.3 Optimization

Based on the findings from above mentioned testing methods, various problems were discovered during the implementation process and as a consequence, optimizations were performed. This includes enhancements within the implemented communication components and conceptual improvements of the individual security protocol mechanisms. Also, it comprises the optimization of the performance of the simulator in terms of speed and memory. The most important optimizations are presented in the following.

6.3.1 Conceptual

Drop/Hold mechanism: If a sender in either of the three presented protocols tries to send a message on the Application Layer, that is not yet authorized, on the first sending, the authorization is performed. After that, all messages that are incoming on the Application Layer need to be handled by the Communication Module. Two options are implemented for this case: Drop and Hold. If the Hold option is active, all incoming messages are stored in a queue of defined length, while the authorization is running. Subsequently, once the stream is authorized all messages in the queue are sent immediately with the specified security mechanism. In contrast to that, when applying the Drop option, all incoming messages are simply dropped and only messages that arrive after the authorization are transmitted. This is a valid option, as messages can lose their validity after the authorization has passed.

Discard irrelevant messages: In order to reduce the number of SimPy events in the LASAN protocol, the SM accepts only one Request Message per stream. All other incoming messages are dropped. Also, if the ECU is already authenticated the Confirmation Message is ignored.

Request repetition: Another problem in LASAN that causes major traffic on the bus is the following. Messages are sent in time intervals that are between 0.001 and 1 second by the Application Layer. Thus, while the stream is not authorized, the Communication Module sends one Request Message on each incoming message from the Application Layer. Consequently if this is done by 100 ECUs that send 500 messages, this results in an overload of the bus, in a full SM receiving buffer and in a huge number of events that slow down the simulator. That is why a minimum interval is introduced, in which the ECU is allowed to send a Request Message. This ensures that the event number is minimized, as well as that the SM and the bus are not overloaded.

Deny Messages: A conceptual improvement that is introduced for LASAN is the Deny Message. It is possible, that an ECU requests a stream that it is not allowed to send. Thus, when sending a Request Message, the ECU waits for the granting of the stream. Consequently, if it is not allowed to send the stream, it would wait and resend the request. That is why the SM is answering with a Deny Message, that ensures that the requesting ECU is informed about the negative outcome of its request.

6.3.2 Speed

The main factors that influence the performance of the simulation in terms of execution time are the number of events that are created in the SimPy environment, the existence of redundancies in the code, the data structures that are created and the abstraction level of mechanisms in the implemented security protocols. That is why, to optimize the speed of the simulator, the following communication system and protocol specific adjustments are realized.

Drop mechanism: The Hold mechanism, that is described in Section 6.3.1, leads to additional slowing processes that are necessary to manage the holding of messages. On top of that, when Hold is set, messages are stored until the authorization is complete. Thus, once this happens all messages in the buffer are sent at once. Consequently, in the smallest possible amount of time the transmit buffer is filled with those messages. As those messages are to be sent immediately, the bus is busy until all messages are processed. This results in a high number of events in a small amount of simulation time causing further deceleration of the simulators execution time. Consequently, the Drop mechanism is introduced. With this option, messages that are incoming from the Application Layer are dropped if authorization is not finished yet. This is valid as held messages in automotive communication systems are most likely deprecated by the time that they would have been released.

Hardware filter: When ECUs receive messages, they pass them through the Physical, Data Link and the Transport Layer until they reach the Communication Layer. Thus, each received message causes events along those levels in each ECU. To minimize those events in each ECU, only the messages with authorized identifiers are let pass at the Physical Layer and all other layers. In the same way, gateways only receive messages with identifiers that are accepted by them and also only send the ones that are specified. This decreases the number of events and thus, speeds up the simulator.

Disable buffer control: When a message arrives or before it is sent, it has to be checked if the transmit or receive buffer is able to accommodate it. That is why the current size of the messages in the buffers has to be determined. This process includes iteration over the contained messages and thus, redundancy. By disabling this control mechanism those iterations are skipped and a speedup is achieved. This configuration is optional and can be used as an abstraction if the buffer control is not considered and thus, it is assumed to be sufficiently large to not overflow.

Message reduction: A higher number of messages that is sent on the Application Layer leads to more events in the SimPy event list. Thus, in the tests that are performed in the course of this thesis, upon completion of the authorization only a small number of messages is sent per stream. Messages, that are sent in the authorization have lower message IDs than the messages sent after the authorization. Hence, at the time those messages are in the transmit buffer they are transmitted first and the stream messages later. Also during the arbitration, those messages access the bus first. Consequently, to analyze the setup processes of the ECUs, where only those

authorization messages are of concern, it only makes a minor difference if authorized simple messages are on the bus or not. That is why this message reduction can be assumed on the Application Layer.

Sending phases: As it is described above, the number of messages that are sent should be kept low. For this purpose, each security protocol is divided into phases, that are executed step-by-step. In this way each authorization message is only initiated once. This is described in more detail in Section 6.1.3.

CAN bus optimization: Another factor that decreased the performance is the Standard implementation of the CAN bus. In this approach, that is presented in Section 5.1.2, the Data Link Layer produces many events, such as events for the notifications if the transmit buffer is filled and if its message is sent successfully. Also a back-off time creates an additional event. Those events increase with the number of ECUs. Thus, by implementing the Rapid CAN bus implementation (see Section 5.1.2) those events are eradicated and the bus runs in one central process, with one event per message.

Reduce SimPy timeouts: The number of timeouts is reduced by summarizing multiple subsequent timeouts into one timeout. For example, during the generation of the Registration Message of the LASAN protocol, multiple encryption and hashing times are present. Those are summed up to form one SimPy event. At the same time each Monitor input is saved with the time offset corresponding to the actual time when it would appear if individual timeouts were performed.

Cache database requests: When ECUs request time values according to its configuration, an SQL request is performed to the measurement database. In the presented protocols, often similar messages are sent, that request the same data from the database. That is why once a request is performed in the database, the result value is cached. Hence, on any further request of the same type, the data is directly retrieved from the cache instead of accessing the database again.

Direct result output: The presented Monitor principle also requires many iterations during the gathering and publishing processes, as it is stated in Section 5.5.2. That is why an option is implemented to store data directly into a CSV-file instead of passing the whole Monitor structure. Thus, with this step, further overhead is reduced.

Logging and Exceptions: On top of that, several try-except blocks and logging outputs are removed as they are the cause of further slow downs.

6.3.3 Memory

In addition to speed optimizations, also several measures are taken to keep the memory usage of the simulation low as this simulator is required to support large systems of up to 100 ECUs and 1000 messages.

Memory leaks: In a first step, memory leaks are detected by manually inspecting all dictionaries and lists that are present in the simulation. In particular it is ensured, that they are all cleared after they are not needed any more. This leak detection is done by using the tools *Memory Profiler* [Ped15] and *Object Graphs* [Ged15].

TLS optimizations: To optimize the memory usage of the TLS protocol implementation, the following changes are made. Each sender and each receiver needs to create an own certificate with numerous root certificates, that are exchanged. Memory usage is decreased by reusing one single certificate and one group of root certificates for all processes in the simulation. Moreover, in an optional version of the Communication Module the Record Layer is abstracted by not wrapping messages into Plaintext, CompressedMessage and EncryptedMessage objects. Instead a clear list is sent and parameters in the list specify if either of the encryption mechanisms is applied. Those changes are only feasible if solely the timing aspect of the protocol is analyzed. If behavior needs to be investigated this is done with an optional implementation of the TLS protocol.

TESLA optimizations: Each stream that is generated in TESLA requires a defined number of keys to be generated. In order to not repeatedly perform this operation, only one key chain is created per simulation run. This chain is then used for all streams in the simulation. On top of that, the validity of the key chain depends on the starting point of the key with interval index 0. In order to be able to create less keys, the interval 0 begins in the moment the first message is sent. Less keys again result in less memory. This is an valid abstraction, if only temporal aspects are investigated. If behavior is analyzed, this option is disabled.

In this chapter a test case generator is presented that can be used to generate random architectures with random component configurations. This generator is then used to optimize the performance in terms of concept, speed and memory.
On top of that in this chapter four test sets are presented. Those are used to validate the functioning of the simulator as a software. Thus, in the last two chapters all components of the simulator and its validation are presented. Next, in order to show a realistic use case for the simulator, it is integrated with an existing simulator for BMSs. This is explained in the following chapter.

7

Case-Study - Battery Management Simulation

In order to demonstrate a realistic application for the implemented IVNS, in this chapter a case-study is presented. Distributed BMSs, consisting of smart cells, as they are presented in [SLN+14] are simulated with the IVNS, which is described in Section 5.1. For this purpose, an adapter is built that connects both the existing BMS Cyber-Physical Co-Simulation Framework (CPCSF) as it is presented in [Mee15] and the IVNS introduced here.
This is done because the Communication Layer implemented in the CPCSF is focused on functionality and is thus, kept minimal. It assumes all smart cells to be on one bus that is realized as a priority queue. Thus, by exchanging this Communication Layer and adapting it to the components provided by the IVNS a more realistic model is added to this simulation. Consequently, all analysis tools that are presented in the simulation and its modularity becomes available to the CPCSF. Hence, it is possible to arrange the smart cells in any architecture that is available with the presented simulator, using any modules that are implemented for each unit. Moreover, all security protocols can be applied in this framework.
By performing this case-study a realistic application for the simulator is presented. At the same time, the proposed security protocols are analyzed in a promising future sub-network of automotive networks. In the following, in Section 7.1 the BMS that is proposed by [SLN+14] is discussed and the existing CPCSF is introduced in Section 7.2. Next, in Section 7.3 the integration of the IVNS to the present CPCSF is described.

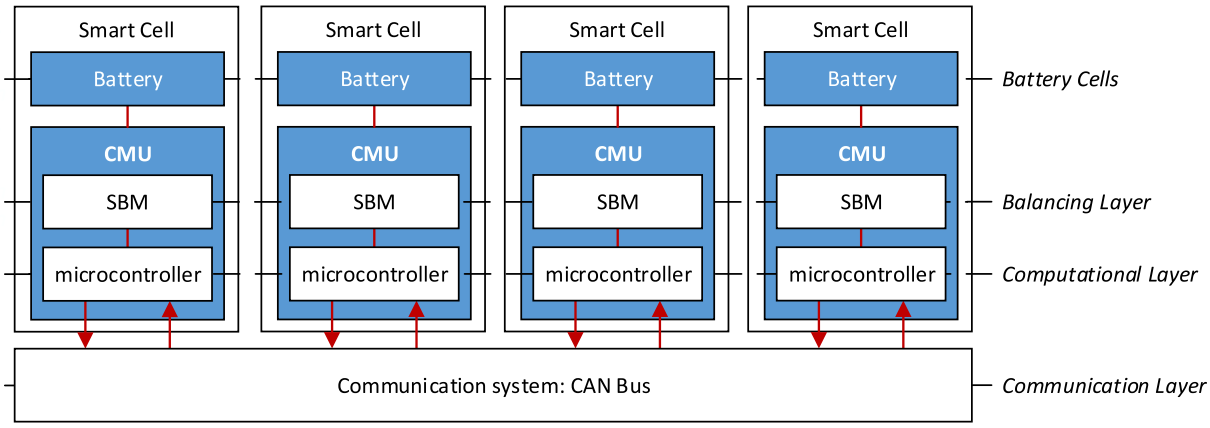

Figure 7.1: Schematic of distributed smart cell BMS. Adapted from [SKM+15].

7.1 Distributed Battery Management System

This section presents the BMS as it is introduced in [SLN+14]. In the continuous trend towards electric vehicles and smart energy grids, Lithium-Ion battery packs, consisting of up to 96 cells, are used to store energy. Throughout their life time, each of those cells loses capacity and thus, fails faster. This is due to manufacturing differences, as well as due to cell voltages and operating temperatures that exceed a reasonable state during charging and discharging of the battery. This state changes in each cell individually as variations in manufacturing lead to different operating temperature per cell. Hence, in order to prolong the life time of the battery pack, in [SLN+14] it is proposed to equip each of the battery cells with a computational unit called the Cell Management Unit (CMU). Those components are responsible to maintain the battery's state of health. This is done by estimating the State of Charge (SoC) of each cell from measured voltage and temperature values and managing it to keep this value in a specified range.
For this purpose, the BMS that is proposed in [SLN+14], is composed of battery cells which comprise a CMU, a Sensor and Balancing Module (SBM) and a transceiver connected via a communication network. This is shown in Figure 7.1. In this approach, multiple of those cells, called smart cells, are interconnected via a CAN bus. In order to keep the SoC value in a defined range, the cells communicate with each other and exchange charge if necessary, causing changes in the cells' SoCs. This is done in a distributed fashion. Thus, every cell manages itself in order to achieve a healthy SoC range. The charge negotiation and exchange process is illustrated on Figure 7.2 and is performed as follows.

1. *SoC broadcast:* With a period of 100 Hz, all smart cells on the bus broadcast a *SoC-Broadcast-Message*. By doing this, all cells on the bus are kept aware of the SoC state of the other cells. Depending on the strategy that is used, a cell can consequently decide to send charges and multiple receivers can decide to receive charge.

2. *Balancing transaction:* Exchange of charge is then initialized by a charge receiving party. It sends a *Send-Request-Message* to the unit that is to release charge. After the charge

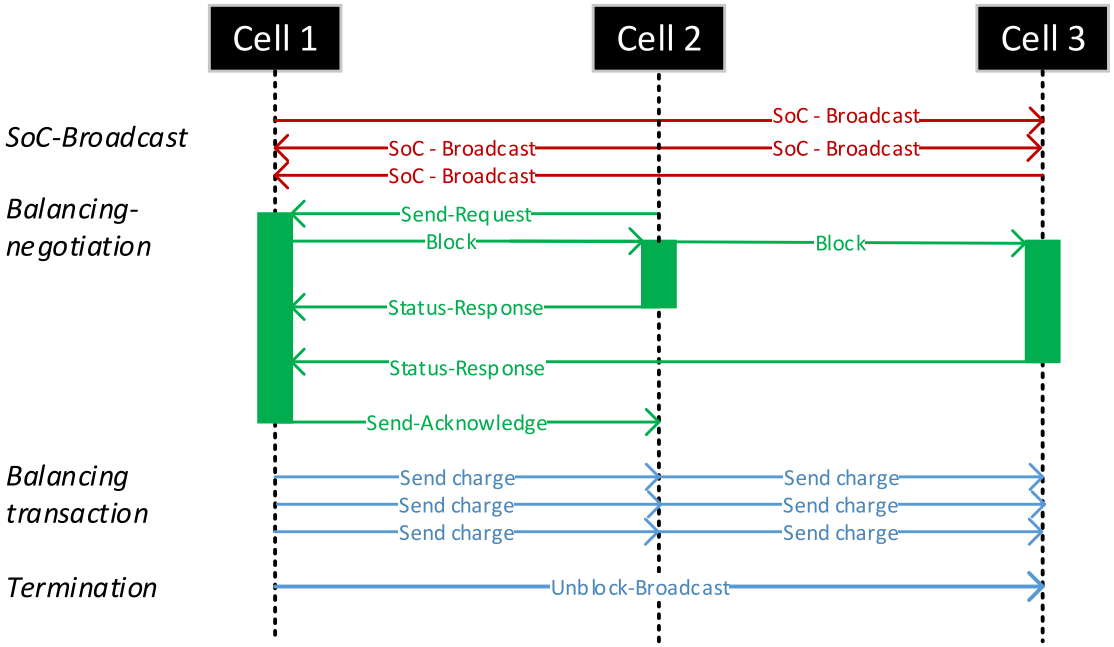

Figure 7.2: Message sequence chart for balancing transaction negotiation in a distributed BMS. Adapted from [Mee15].

sender received the message, a *Block-Message* is broadcasted to all cells on the bus to inform them that the transmission of data is blocked until an *Unblock-Message* is sent. Cells that are to receive charge respond to the *Block-Message* with a *Status-Response-Message* informing the sender that they are ready to receive charge. After the sender got all of those messages, also the requesting party is notified by sending a *Send-Acknowledge-Message* to it. Next, the transmission of charges from the sending unit to all receivers is performed via an additional cable connection between all battery cells. Once the transmission is complete, the sending unit is sending an *Unblock-Message* and the communication continues with the SoC broadcast.

7.2 Battery Management Co-Simulation Framework

After the principle of the distributed BMS is introduced, next, the simulator that implements this system is presented in the following.
In order to test different architectures under various balancing strategies in [Mee15], a simulator is introduced that simulates the behavior of the distributed BMS. Similar to the physical arrangement, the implementation consists of SmartCell objects, including a CMU object, that run in an own SimPy process. On top of that, a global CANBus object is present. All smart cells contain a reference to the global bus object. Once any CMU object calls the send method on the bus, a message is passed and the sending is performed with the timeout calculated from

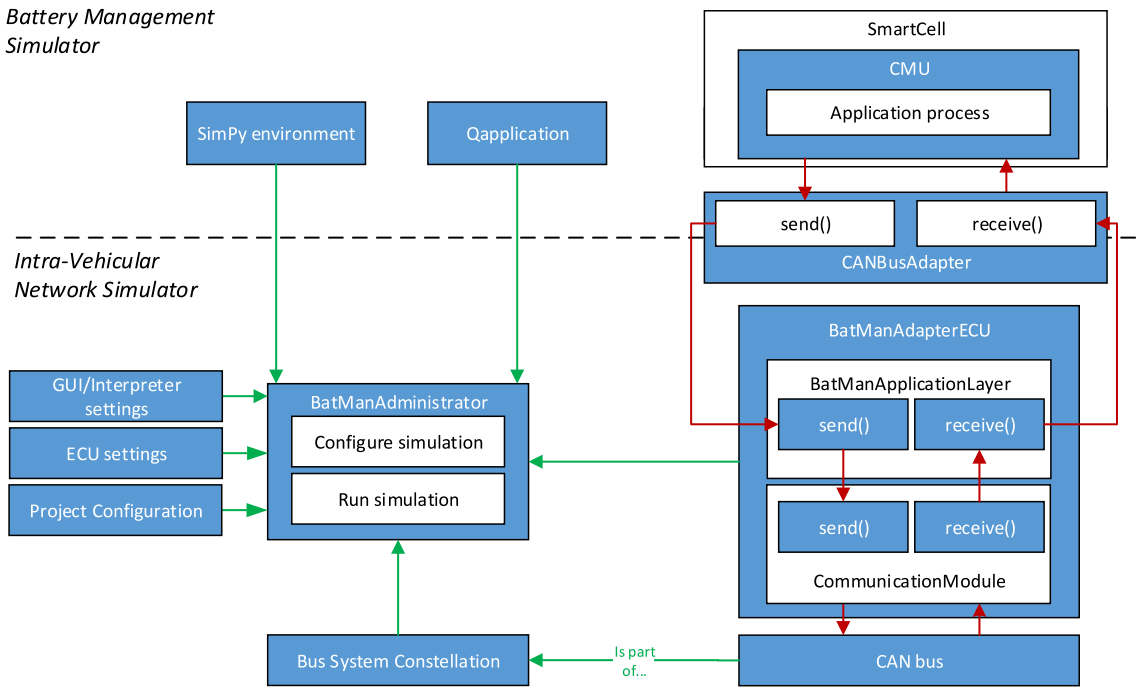

Figure 7.3: Adapter integration simplified.

the message size. Subsequently, a receive method is called that calls the put method of all connected ECUs. This makes the units receive this message.
Each CMU starts with an initial SoC, that decreases according to a defined mathematical model that is described in [Mee15]. Then the balancing described in Section 7.1 is started in each CMU processes. Thus, each process periodically times out for a certain simulation time and sends a *Broadcast-Message*. Upon reception, the request sending condition is checked and a transaction process is initialized if required.
In addition to those CMU processes, a Monitor process is running in parallel that requests the current SoC state from each cell. When requested, this cell state is forwarded to the connected GUI. Similar to the real system, this simulation can be separated into a Communication and an Application Layer. The global CAN bus is accessed exclusively over its send method that invokes the receivers receive method. This method centrally distributes the messages to the receivers and thus, forms the Communication Layer. Contrary, the CMU processes form the application part of the BMS.

7.3 Battery Management - Adapter Integration

After the battery management simulator was introduced in Section 7.2, in the following the integration of this simulator with the IVNS is described. The integration of the IVNS is performed non-intrusive (see Figure 7.3). Thus, the CPCSF can still be invoked as expected. Only if the IVNS explicitly activates its services in the CPCSF, the extension is applied. This is implemented as follows. On the IVNS side, the BatManAdministrator and the BatManCANBusAdapter are introduced. The former is a singleton, that is on the one hand used to modify

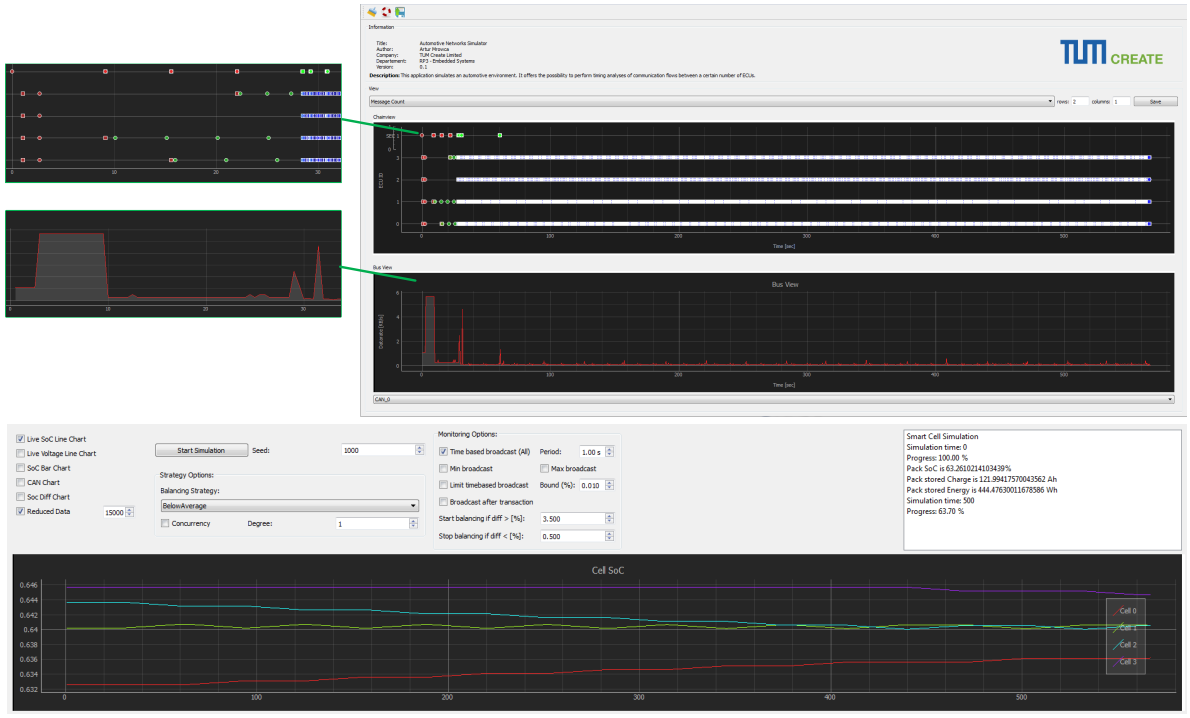

Figure 7.4: Screenshot of the GUI when the simulation is running. *top:* Intra-vehicular network simulation. *bottom:* Battery management simulation.

the architecture and configure the setup on the IVNS side. On top of that, the BatManCANBusAdapter is used to connect to each CMU of the CPCSF.
In this integration CPCSF is treated as Application Layer from the perspective of the IVNS. Vice-versa the IVNS is treated as CAN bus in the perspective of CPCSF. The integration of both sides and the processes to start the simulation are presented in the following.

CPCSF side adjustment: Using a tag, the CPCSF is aware if it is started with or without the IVNS. Thus, if the simulator integration is active, three adjustments are made on the CPCSF side. First, the GUI QApplication object is stored to the BatManAdministrator Singleton [Bis07]. Next, the SimPy environment that corresponds to the configured CPCSF is passed to the BatManAdministrator. Lastly, each CMU keeps a reference to a global CAN bus object, that distributes the stream to all connected receivers. This is done by pushing it to a central send method and receiving it in a central receive method. At this point the simulator hooks in. Thus, in each CMU object the central CAN bus object is replaced with a BatManCANBusAdapter object. Each of those objects represents the connection to one CMU in the IVNS network simulation.

Intra-Vehicular Simulator side adjustment: Via the BatManCANBusAdapter each CMU of the CPCSF is mapped to one BatManAdapterECU that has a BatManApplicationLayer, which handles the CANsend and CANreceive method of the CMU. Thus, each send method called by the CMU invokes the sending method of the corresponding BatManApplicationLayer.

Analogously, once the BatManApplicationLayer receives a message it pushes it to the CMU receive method via the connected BatManCANBusAdapter. So the BatManCANBusAdapter is the connection point between both simulators, as it is instantiated in the CPCSF and connects the sending and receiving methods to the IVNS. On top of that, the BatManAdministrator is introduced on this side. It is used to configure the project via the API as described in Section 5.4. Next, it establishes the mapping from CMUs to BatManAdapterECUs and starts the simulation by running the SimPy environment. In particular, the BatManAdapterECU is able to apply any communication protocol that is specified within the IVNS. Thus, the Communication Module can be adjusted as desired and the behavior of the CPCSF can be analyzed if any security protocol is applied.

Starting simulation: To start a simulation, first the BatManAdminitrator and the API are used to configure the project. Thus, on the side of the IVNS all adjustments that are discussed in Chapter 5 are set. At the same time all application options of CPCSF are set on side of CPCSF. The BatManAdminitrator is then used to launch the environment.

A screenshot of both simulations running in parallel is pictured in Figure 7.4. In this chapter the integration of the CPCSF that is presented in [SLN+14] with the IVNS, that is proposed in this thesis, is discussed. For this purpose, the distributed BMS is introduced, together with its CPCSF simulator according to [Mee15]. Lastly, the integration of CPCSF and the IVNS is explained. In the next chapter, an evaluation of all measurements that are performed on the STM32 microcontroller and the performance evaluation of the simulator are presented.

8 Evaluation

In this chapter the outcome of measurements that were performed within the scope of this thesis is evaluated. This includes the results of the measurements that were performed to obtain the times for cryptographic operations in the way that it is described in Chapter 4. This also comprises the results measured for the performance of the simulator presented in Chapter 5. That is done using the test case generator described in Section 6.1. Lastly, further performance indicators were determined for the proposed simulator when it is integrated with the CPCSF as it is described in Chapter 7. This is presented in the following.

8.1 Evaluation of Cryptographic Measurements

With the methods described in Section 4.2, several measurements are taken on the evaluation board. The results of those measured times are discussed in the following. First, for each algorithm, shortly the experiment is presented. Next, the results are described and its interpretation is discussed. In the following those algorithms are described sequentially.

8.1.1 Hashing Algorithms

By applying the algorithm that is described in Section 4.2, the times for cryptographic operations were measured. Those were measurements of algorithms' processing times using a STM32 microcontroller. The values that were measured for MD5, SHA1, SHA256 with STM Cryptographic Library (STMCL) or CyaSSL are shown in Table 8.1 and are discussed in the following. In this table the entry SW indicates that only software values were determined, while SW and HW indicates that also hardware values were measured for the respective entry.

Library	Hashing	Mode
CyaSSL	SW	MD5
CyaSSL	SW	SHA1
CyaSSL	SW	SHA256
STM Cryp. Lib.	SW	MD5
STM Cryp. Lib.	SW	SHA1
STM Cryp. Lib.	SW	SHA256

Table 8.1: Overview of measured timing values for hashing algorithms.

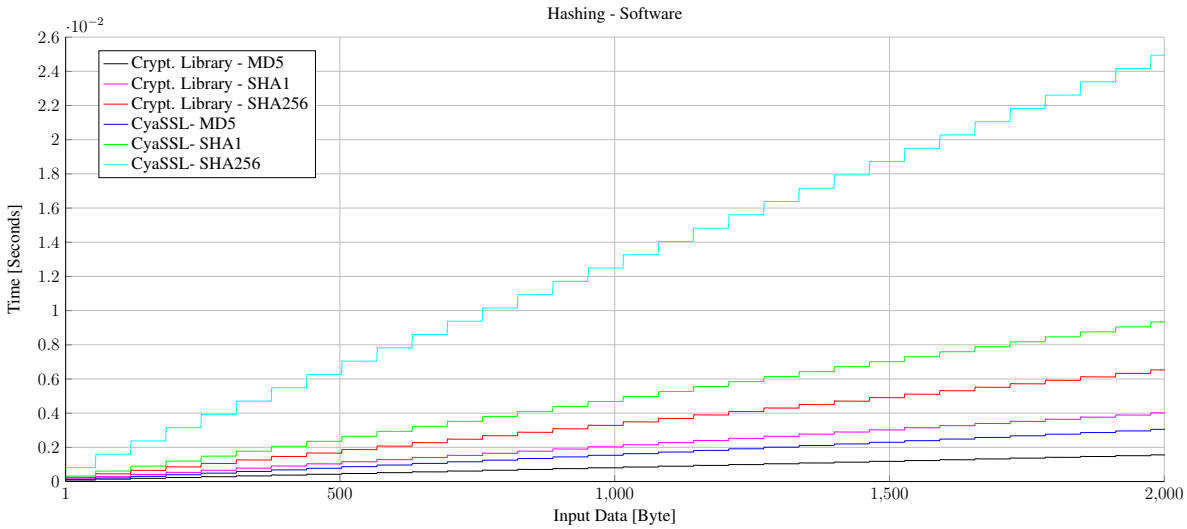

Figure 8.1: Resulting hashing times for different input data lengths depending on the applied library.

Results: As it is illustrated in Figure 8.1, for all hashing algorithms the hashing time increases, when more bytes are processed. Thus, for MD5 values are between 95 μs and 3814 μs for hashing of 1 to 2500 bytes. For SHA1, those are between 165 μs and 11664 μs, as well as for SHA256 between 251 μs and 31170 μs. For those algorithms in regular steps of 64 bytes the processing time increases. Also STMCL gives smaller values than its CyaSSL counterpart.

Interpretation: When comparing hashing algorithms, MD5 performs best and the STMCL implementation requires less time than the one of CyaSSL. The next fastest processing times result from SHA1 followed by SHA256 on the STMCL. The longest hashing times result from the CyaSSL implementation of SHA1 and SHA256.
MD5 is faster than the other algorithms as it performs the smallest amount of rounds and steps as it is described in Section 2.3.2. That is also why SHA1 performs faster than SHA256. The STMCL implementation is more efficient, as their processing times for all algorithms are faster. On top of that, steps are present in all curves, because the algorithms process message blocks of 512 bits length and padding is applied. Padding means that data bytes are filled up with zeros

Library	Enc.	Dec.	Key gen.	Mode	Key lengths
CyaSSL	SW	SW	SW	CBC	128, 192, 256
CyaSSL	SW	SW	SW	CCM	128, 192, 257
CyaSSL	SW	SW	SW	CTR	128, 192, 258
STM Cryp. Lib.	SW and HW	SW and HW	SW and HW	CBC	128, 192, 256
STM Cryp. Lib.	SW and HW	SW and HW	SW and HW	CMAC	128, 192, 257
STM Cryp. Lib.	SW and HW	SW and HW	SW and HW	CTR	128, 192, 258
STM Cryp. Lib.	SW and HW	SW and HW	SW and HW	ECB	128, 192, 259

Table 8.2: Overview of measured timing values using the AES algorithm.

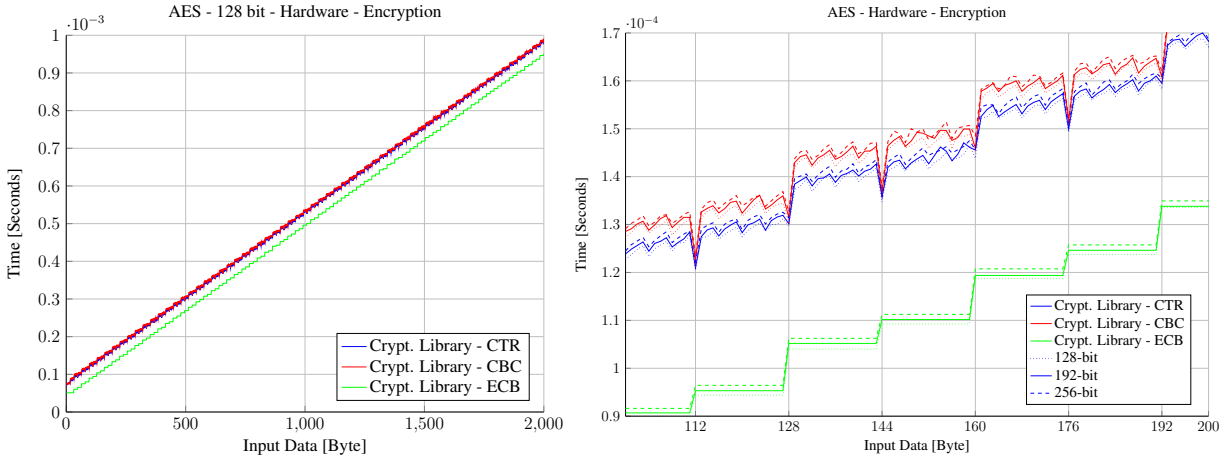

Figure 8.2: AES hardware encryption times for different input data lengths.

until they have a certain block length. Thus, after each 64 bytes, one block more needs to be processed, resulting in steps at those values. Moreover, within those steps, the encryption time is constant as blocks are padded to form 512 bit blocks. Thus, per number of processed blocks, identical processing steps are performed by the algorithm.

8.1.2 AES Algorithm

Similar to the previous section, the values as they are shown in Table 8.2 were measured for AES with STMCL and CyaSSL.

Results: In all results obtained for AES, the times needed to encrypt and decrypt data increases in steps of 16 bytes. If STMCL is used in hardware mode (see Figure 8.2) and AES with 128 bit key size is applied, values are between 0.0509375 ms for 1 encrypted byte and 0.978 ms for 1600 bytes. In software mode, for the STMCL, encryption times are in the range of 0.042 ms and 7.615 ms. For CyaSSL this range is 0.0365 ms to 1.58 ms.
In all modes, the encryption and decryption time also increases when the key size is raised. This increase is around 10^7 s per next higher key length.

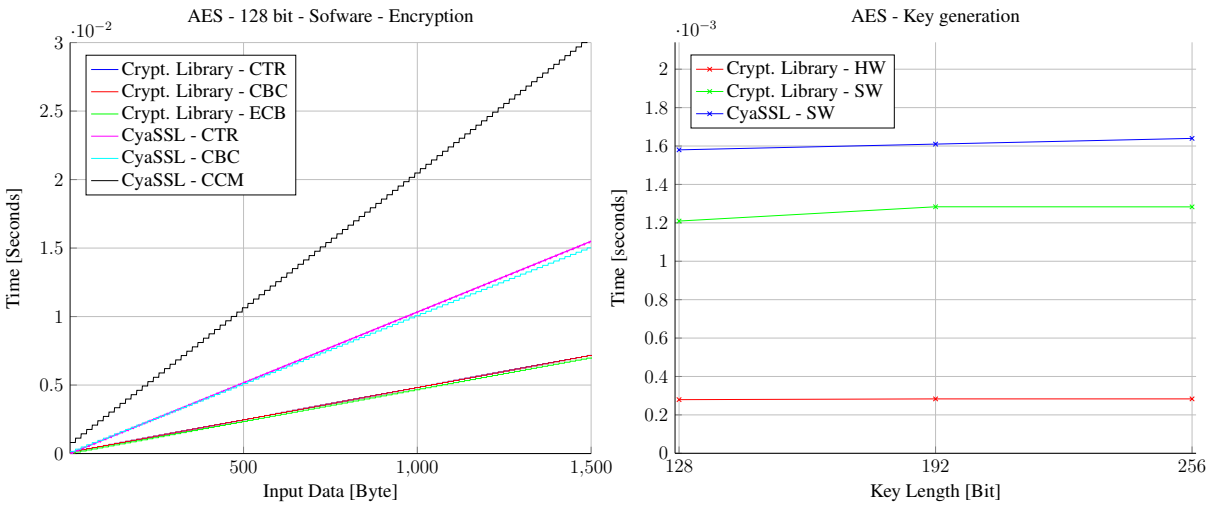

Figure 8.3: *left:* AES software encryption times for different input data lengths. *right:* AES key generation times depending on generated key length.

The processing times for STMCL software (see Figure 8.3) are slower than their hardware counterparts. The deviation of those increases with the number of encrypted bytes. This difference ranges from 20 μs for 1 encrypted byte to about 7 ms if 1600 bytes are encrypted.
As it is pictured in Figure 8.3, the random number generation for 128 bit long keys with STMCL in hardware mode takes 0.279 ms and 1.209 ms in software mode. CyaSSL for the same task requires 1.580 ms. Respectively, the times increases with the length of the created key.

Interpretation: AES is processed in 16 byte blocks and padded to a multiple of it, if smaller sizes are passed to the algorithm. Thus, on the AES curves steps are present in distances of 16 bytes as more blocks are processed after each of those steps. Consequently, there is a linear deviation between all curves resulting in an increasing gap when input data sizes rise. Each block is processed in different processing times for the individual algorithms. Hence, more blocks mean multiple constant deviations resulting in a growing gap.
On top of that, ECB mode performs fastest as its implementation processes all blocks individually without the involvement of prior calculations, while both CBC and CTR mode perform additional calculations on prior cipher texts and initialization vectors. For this reason those modes require more processing steps and thus, more time for encryption and decryption.
On top of that, the hardware implementation is faster. This is because in those implementations, the calculations are outsourced to further units that perform a parallelized encryption.
Random number generation is also done in multiple rounds. Thus, to generate more random bytes more steps are needed. That is why the generation of 128 bit keys is processed faster than the generation of 192 bit and 256 bit keys. Moreover, again the hardware implementations execute this task parallelized and are therefore faster than their software counterpart.
For AES, random number generation and hashing, all algorithms perform faster with the STMCL. This is due to the fact that the manufacturer of this library does know the chip architecture and

Library	Alg.	Enc.	Dec.	Sign	Verify	Key g.	K./P. len.	Expo.
CyaSSL	RSA	SW	SW	-	-	SW	512, 1024, 2048	3, 5, 17, 257, 65537
STM Cryp.. Lib.	RSA	-	-	SW	SW	-	512, 1024, 2048	3, 5, 17, 257, 65537
CyaSSL	ECC	SW	SW	-	-	SW	256, 384, 521	
STM Cryp.. Lib.	ECC	-	-	SW	SW	SW	192, 256, 384	

Table 8.3: Overview of measured timing values for asymmetric encryption algorithms

thus, is able to implement a faster implementation as both the library and the chip are developed by ST Microelectronics.

8.1.3 RSA Algorithm

Also for RSA, measurements were taken as described in Section 4.2. The performed measurements can be seen on Table 8.3. In the following their outcomes are discussed.

Results: For RSA (see Figure 8.4), data is padded to be of a size $s = keylength - 11$ byte. Thus, for 512 bit keys 53 bytes, for 1024 bit keys 117 bytes and for 2048 bit keys 245 byte were measured. With an exponent of 65537 and a key length of 512 bit, the signing procedure in STMCL takes 0.262 s, while the verification takes 0.0303 s. This remains constant throughout the measured input data size. Both times increase with the key length. Hence, for 1024 bit signing takes 1.711 s and verification requires 0.116 s. In case of 2048 bit those times are 12.255 s and 0.368 s respectively. Signing and verification times also vary within a range of 0.1 s with the applied exponent.
The public encryption times measured with CyaSSL for 512 bit keys and exponents of 65537, require 0.205 s. Similarly, the private decryption times are around 0.226 s. Both latencies increase with bigger key sizes. Additionally bigger exponents increase the encryption time, while the decryption time remains constant.

Interpretation: In RSA signing and verification operations are independent of their input data length. This is because any data that is smaller than the used key length is padded to the size of the key. Thus, as shown in Figure 8.4, those operations are nearly constant for fixed key sizes. On top of that, the signing operation takes longer than the verification as the private exponent is chosen to be big in comparison to the public key. For this reason, in STMCL for an exponent of 65537 and a 512 bit key, the signing time is nearly equal to the verification time. For 1024 bit keys it is about 15 times higher than the verification and for 2048 bit keys it is about 33 times higher. The same behavior can be seen on the results for the CyaSSL library. For the same reasons, the public encryption time of CyaSSL is lower than its private

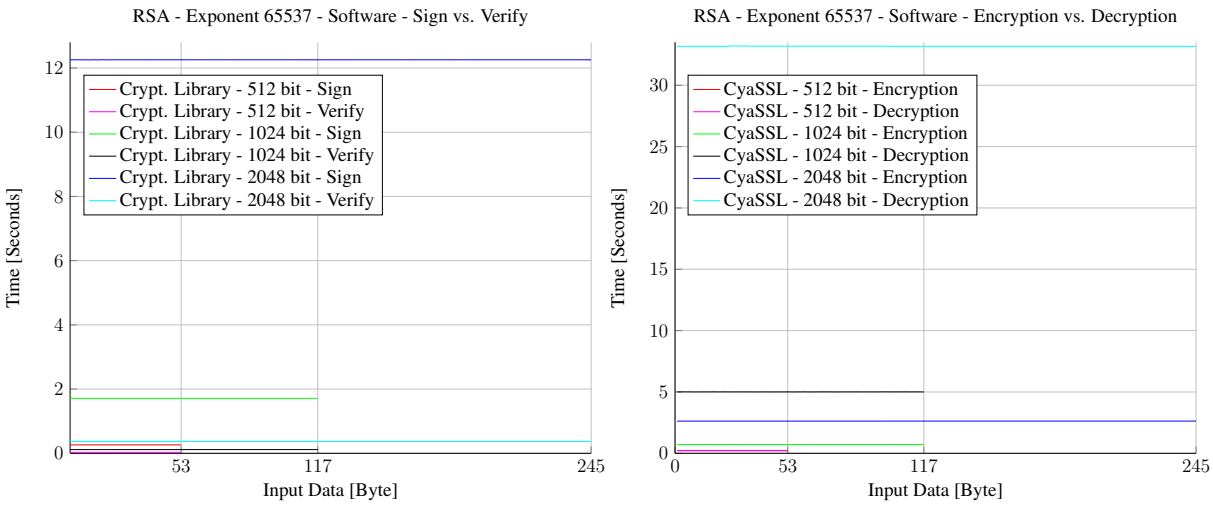

Figure 8.4: *left:* RSA signing and verification procedures with varying key lengths. *right:* RSA public encryption and private decryption with diverse key sizes.

decryption. Nevertheless, in all RSA measurements the processing times increase when bigger keys are used. This is due to the fact that in this case more data is processed at once.

8.1.4 ECC Algorithm

For ECC, similar to the previous cases, measurements as introduced in Section 4.2 are performed. All measured parameter sets are shown on Table 8.3.

Results: For both libraries signing and verification operations with ECC, as it is plotted in Figure 8.5, remain constant per parameter size (= key size). With STMCL this operation takes 0.401 s for parameter lengths of 192 bit. Similarly, the verification time is around values of 0.772 s for 192 bit parameter length. Both times increase with higher parameter lengths. If the same procedure is performed with CyaSSL, this operation takes around 3.757 s for 256 bit keys and also increases with the parameter size.

Interpretation: For both libraries, the ECC signing operation is faster than their verification counterparts when the same parameter length is used. This is because, the signing procedure is more costly. Additionally, longer parameter sizes require more time for signing and verification. This is because bigger amounts of data are processed in this case.
Public encryption is slower than the private decryption within same parameter lengths as the former operation is more complex and thus, more costly. In addition to that, bigger parameter lengths lead to longer encryption and decryption times. The reason for this is that more data is processed in ECC for bigger parameter lengths and thus, adds cost to the calculations. This results in longer processing times. Lastly, in ECC data is padded to a fixed size before the algorithm is applied. Thus, the timings are independent of the input lengths of the data.

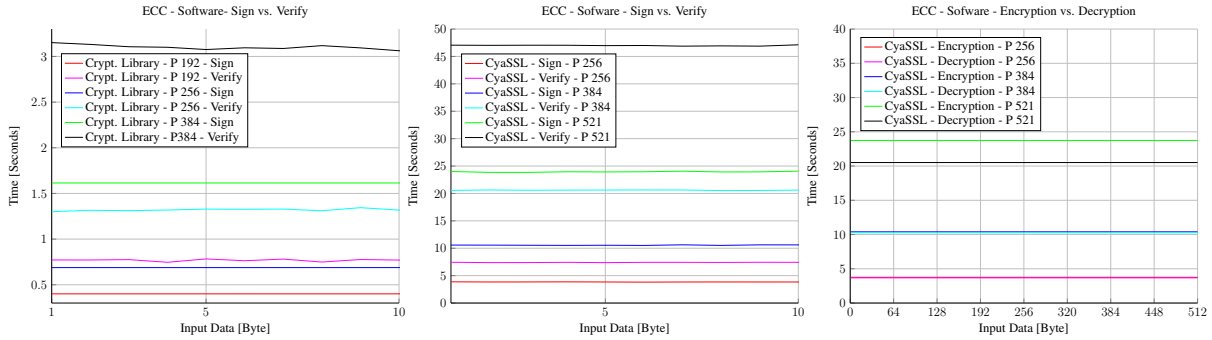

Figure 8.5: *left and middle:* ECC signing and verifying procedures with varying key lengths. *right:* ECC public encryption and private decryption with diverse key sizes.

So far the evaluation of the processing times of cryptography is discussed. This includes measurements for hashing, encryption and decryption, as well as for signing and verification on the STM32 microcontroller. The measured data is used within the simulator that is proposed in this thesis. This simulator is evaluated in terms of performance, which is described in the next section.

8.2 Simulator - Performance

In this section the performance of the IVNS is evaluated using the test case generator that is proposed in Section 6.1. The run time of the simulation, the memory usage and the ratio of computation time to simulation time are used as performance indicators. In the following first the experiment is described in Section 8.2.1. After that, results and their interpretation are discussed for the Rapid mode in Section 8.2.2 and for the Standard mode in Section 8.2.3. Lastly, in Section 8.2.4 the same performance indicators are discussed that occur in the IVNS when it is integrated with the CPCSF simulator for the BMS.

8.2.1 Experiment

All experiments were executed on a 64-bit Windows 7 computer, with 4 GB of RAM and an Intel Core i5-3450 CPU Quad-Core processor at 3.1 GHz. The test case generator is started for all three protocols. It is executed multiple times within a range of 3 to 102 ECUs in steps of 3. Also the number of sent messages is varied between 10 and 1000 with a step size of 30. On top of that, tests are performed with 1, 3, 5, 7 and 9 buses. The random factors of the test case generator are adjusted as follows. Each time that 5 ECUs are added to the system another receiver is added to a message stream. For 2 ECUs each message stream is sent to one receiver. (see Section 6.1)
In all of those test cases CyaSSL was used as library for timing dependencies and all asymmetric algorithms are set to RSA, with an exponent of 65537 and a key length of 512 bit. The symmetric algorithms are set to AES in ECB mode and use key sizes of 128 bit. For tests with

the TESLA protocol 400 000 keys are created. For all protocols after the setup phase, five messages are sent until the simulation is stopped.
Those measurements are performed in Rapid mode with all optimizations described in Section 6.3. Additionally, the behavior of the Standard mode is determined with various configurations. It is started with 69 ECUs and 10 to 730 messages and with 730 messages and 3 to 81 ECUs. In both the Rapid and Standard mode one bus is used. Those measurements are performed with a Monitor. This Monitor samples and publishes once in five seconds of simulation time. This is done in order to demonstrate the effect of the optimizations that were described. The discussed curves show mostly 310 messages, as this resembles the amount of message streams in low cost cars, while 730 of them are representative for middle class cars. For TLS, only measurements of up to 21 ECUs and 190 messages were performed as the simulation execution time falls out of a reasonable range. This reasonable range is defined to be no more than 8 hours of execution time. On top of that, the term computation time is the time the simulation needs to finish its execution in the real world, while the term simulation time refers to the time that passes in the simulation.

8.2.2 Rapid Mode

In this section the results for the measurements performed in Rapid mode are discussed that were achieved with the experiments described in Section 8.2.1. For this purpose the outcomes are discussed for the three cases when the ECU, message or bus number is increased in the experiment.

Increase number of ECUs

Results: As can be seen in Figure 8.6, the measured time needed to execute the simulation increases with an increasing slope if ECUs are added to the system for all protocols. On top of that, the TLS protocol performs slowest, while the time for simulation is nearly equal for TESLA and LASAN. Moreover, the slope of the curves is higher, when more messages are sent. Thus, for TLS, the execution time for 3 ECUs and 730 messages is 28.73 s and for 102 ECUs it is 4647.89 s. For LASAN it is 11.68 s and 842.95 s, while for TESLA it is 9.79 s and 990.87 s.
The ratio of computation time to simulation time rises with an increasing slope for all three protocols. This slope is steepest for TLS, followed by TESLA and TLS (see Figure 8.6). For 730 messages, TLS has a ratio of 0.027 for 3 ECUs and 6.81 for 102 ECUs. Similarly TESLA has ratios between 0.00018 and 0.11, as well as LASAN has ratios from 1.05 to 4.91.
The average memory usage behaves differently for the three protocols, as it is illustrated in Figure 8.6. While for LASAN and TESLA it is nearly constant when ECUs are added to the system, for TLS it is rising linearly. Thus, with 730 messages in LASAN it remains around 110 MB and for TESLA it is around 230 MB. With 730 messages, for TLS the average memory usage ranges from 124.19 MB for 3 ECUs to 589.27 MB for 102 ECUs.

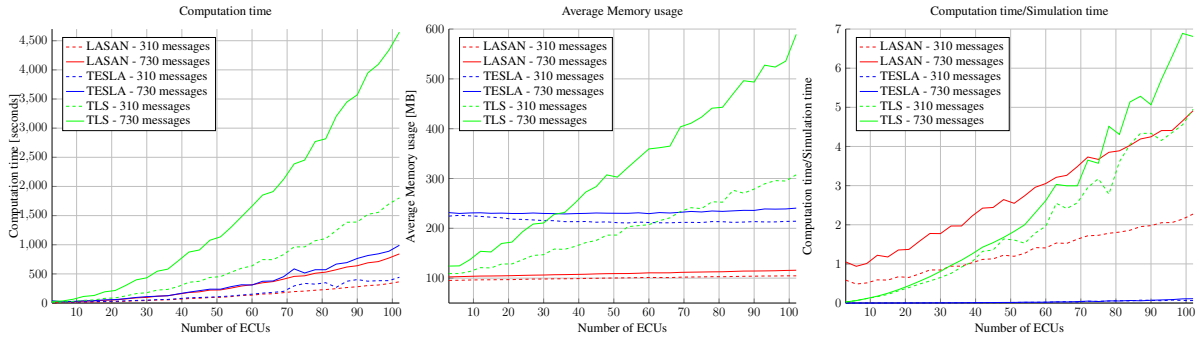

Figure 8.6: Rapid mode with one bus. Computation time, average memory usage and computation time to simulation time plotted against number of ECUs.

Interpretation: Increasing the number of ECUs increases the number of SimPy process, as well as the number of senders and receivers according to the definition of Section 6.1.2.
For LASAN, each additional ECU leads to another ECU Authentication that is performed and an additional receiving process that is performed for each message that this ECU receives. Hence, this results in a curve with an increasing slope for the execution time of the simulation. Similarly, in TLS and TESLA, additional processes are required, as well as additional mechanisms at the receivers, including Handshakes and key exchanges. This results in an increasing polynomial curve. TLS requires the highest amount of additional processes and messages resulting in the fastest growth of this curve.
The ratio of computation time to simulation time for LASAN rises as each additional ECU brings an additional ECU Authentication and multiple Stream Authorization events, which together require less simulation time. Thus, the computation time resulting from more processes increases faster than the addition of simulation time. This leads to an increasing curve (see Figure 8.6). The same holds for TESLA. The additional event processing times grow faster than the additional short lasting message exchanges in the simulation processes. In TLS however, this curve is more distinct, as each additional Handshake requires multiple messages that are sent, resulting in a long computation time. The simulation time that passes with each new Handshake and ECU can not make up for the required time to calculate those, resulting in an curve with increasing slope.
The average memory usage nevertheless remains constant for TESLA and TLS. This is because ECUs do not store messages in those simulations. Contrary, TLS needs to cache verification messages. Thus, more receivers store additional references resulting in a linear increase of memory usage when ECUs are added (see Figure 8.6).

Increase number of message streams

Results: If more messages are added to the system, the computation time increases linearly (see Figure 8.7). The slope is steepest for the TLS protocol. Clearly, the curve of the TESLA and LASAN protocol is less steep. Thus, with 69 ECUs for 10 to 1000 messages it ranges from 15.15 s to 548.63 s for LASAN, from 14.14 s to 598.74 s for TESLA and from 26.91 s to

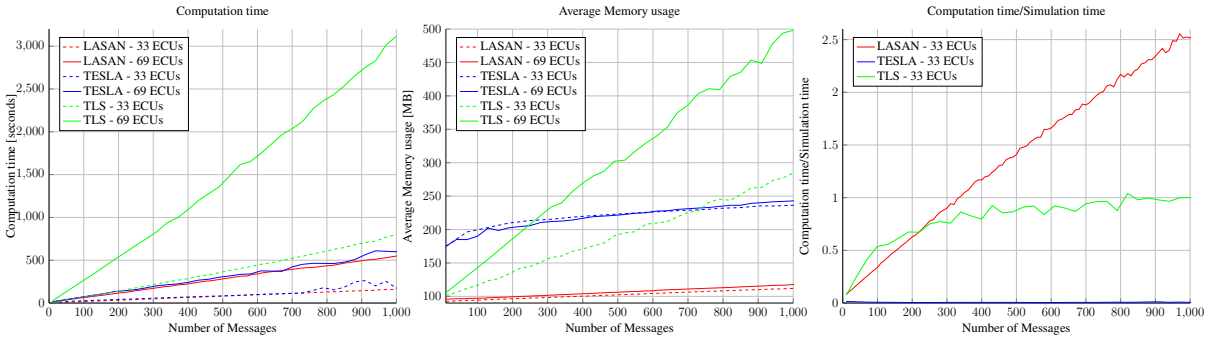

Figure 8.7: Rapid mode with one bus. Computation time, average memory usage and computation time to simulation time plotted against number of messages.

3117.47 s for TLS.
In this case, as shown on Figure 8.7, the ratio of computation time to simulation time increases nearly linearly for LASAN, rises linearly for TESLA and converges with a decreasing slope for TLS towards 1. For the mentioned architecture, ratios are in the range of 0.15 and 4.50 for LASAN, in the range of 0.011 and 0.043 for TESLA and in the range of 0.30 and 4.5 for TLS.
The average memory usage also increases linearly if messages are added (see Figure 8.7). Again the slope is steepest for TLS and less steep for TESLA and TLS. For the range of 30 to 1000 messages and 69 ECUs, the average memory usage for LASAN lies between 95.67 MB and 117.67 MB, for TESLA between 174.92 MB and 242.82 MB, as well as between 105.40 MB and 498.34 MB for TLS.

Interpretation: When messages are added to an environment, per message, a fixed number of additional events in the system is introduced (see Figure 8.7). In LASAN this includes an additional Stream Authorization process with additional five messages that are encrypted and decrypted. Furthermore, in TESLA this is an additional key exchange and also five more messages. TLS performs one additional Handshake per new message stream together with five more messages. That is why the simulation execution time increases linearly for all protocols when new ECUs are added (see Figure 8.7). TLS has the steepest slope as more events result from additional processes per message, as it is the case in TESLA or TLS.
The ratio of computation to simulation time for TESLA remains beneath 0.1. This is because, in the simulation 400 000 keys are generated per message stream. In the computer program, however, those keys are created once and reused as references for all streams. Thus, the computation time remains small and increases only little. For LASAN this ratio is smaller than 1 for up to 310 messages, when 33 ECUs are used. It increases because each new message requires little simulation time as it uses symmetric encryption, while the number of events and processes increases constantly. For TLS this ratio increases steeply at the beginning and converges towards 1 for the performed measurements. This is due to the fact that the computation time rises linearly and the simulation time increases linearly with a different slope.

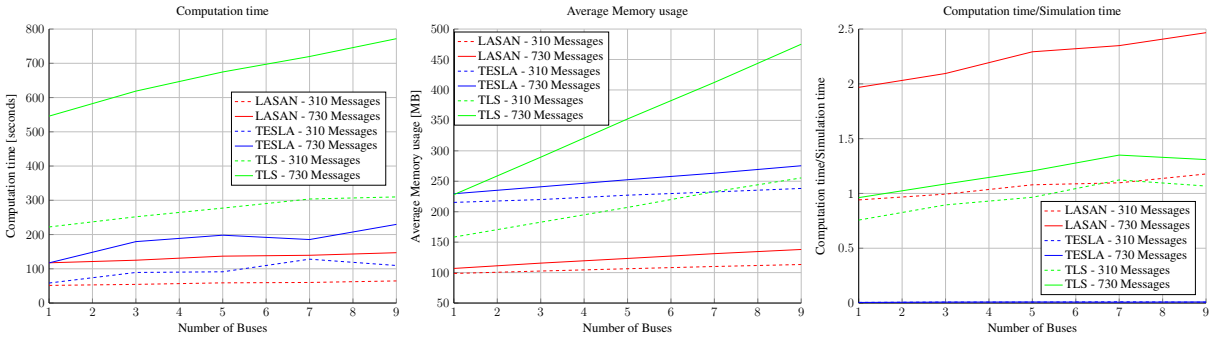

Figure 8.8: Rapid mode with 33 ECUs. computation time, average memory usage and ratio of computation time to simulation time plotted against number of buses.

The average memory usage increases, as each message requires to be saved and more events need to be saved in the event list. Analogously, TLS carries the biggest amount of additional events per message and thus, the steepest slope in this curve.

Increase number of buses

Results: In Figure 8.8 the number of messages is kept at 310 and 730. Also, the number of ECUs is kept constant at 33, which resembles a low cost car. As is illustrated in Figure 8.8, when buses are added to the system, for all protocols an increase in computation time and average memory usage is present. For LASAN, the computation time with 730 messages ranges from 117.46 s for one bus to 146.98 s for 9 buses. In the same way it is between 545.74 s and 771.88 s for TLS, as well as between 117.24 s and 229.65 s for TESLA.
The ratio of computation time to simulation time also increases linearly from 1.97 to 2.47 for LASAN, from 0.0056 to 0.01 for TESLA as well as from 0.96 to 1.31 for TLS.
The average memory usage is between 106.83 MB and 137.98 MB for LASAN, between 229.51 MB and 275.47 MB for TESLA, as well as between 228.07 MB and 475.08 MB for TLS.

Interpretation: Adding buses to the system adds additional events to the environment. The corresponding behavior is illustrated in Figure 8.8. Thus, for each additional bus, transmitted messages are forwarded to each connected bus, which is done in a delayed SimPy process per sent message. Moreover, per additional bus another SimPy process is started that is executed for each sent message. Hence, for each bus that is added, per message multiple SimPy events and additional bus actions are executed. This results in an increase of the computation time and average memory usage. Lastly, the ratio of computation to simulation time rises, as each gateway introduces a small delay in simulation time and the calculation take comparably long.

8.2.3 Standard Mode

For the Standard mode similar behavior is seen as in Rapid mode. Thus, in this section the results are described if ECUs or messages are added to the system. Based on this, an interpretation

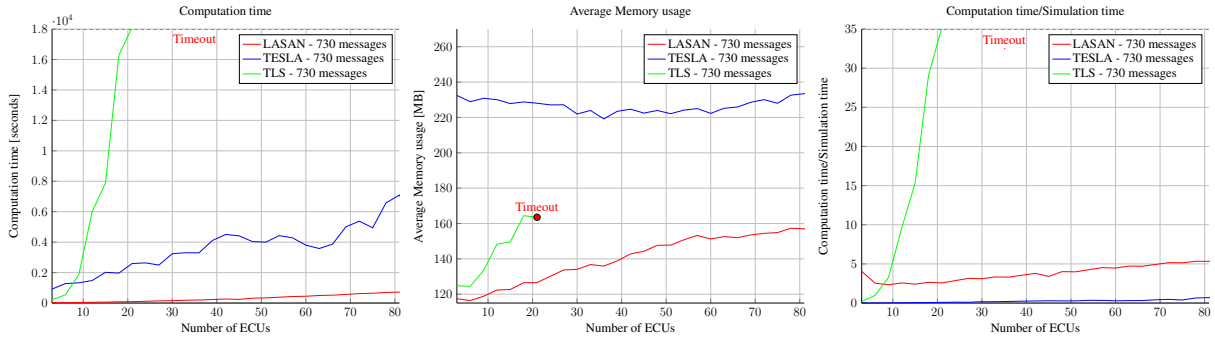

Figure 8.9: Standard mode. Computation time, average memory usage and ratio of computation time to simulation time plotted against number of ECUs.

of this data is presented for each case. This is done by comparing it to the Rapid mode.

Increase number of ECUs

Results: If the ECU number is increased in the Standard mode (see Figure 8.9) similar behavior to the one in Rapid mode is seen. The computation time for TLS, TESLA and LASAN is rising with an increasing slope. In contrast to the Rapid mode, however, for TESLA and LASAN the curvature is smaller, while for TLS it is higher. For 3 to 81 ECUs, values range from 23.68 s to 719.1 s for LASAN and from 918.63 s to 7093.22 s for TESLA. For TLS they are out of the maximum measured range of 8 hours.
The ratio of computation time to simulation time behaves similarly to the Rapid mode. TESLA and LASAN grow linearly, while TLS increases with an increasing slope. For 3 ECUs, LASAN has a ratio of 4.06 and for 81 ECUs it has a ratio of 5.33. In this range the TESLA protocol has ratios between 0.0043 and 0.69.
Similar to the Rapid mode the average memory usage for TESLA remains constant, while the curve for LASAN is increasing linearly in contrast to the Rapid mode. For TLS, the curve increases similarly to the Rapid mode case. Nevertheless, the slope is steeper than in Rapid mode. Thus, for LASAN those values are 117.46 MB for 3 ECUs and 157.02 MB for 81 ECUs, while for TESLA they are around 230 MB for the same ECU range.

Interpretation and comparison: Most curves in Figure 8.9 show similar behavior as in Rapid mode. Nevertheless, the slope and the magnitude of the Standard curves for the considered performance indicators is higher if the ECU number is increased.
For the computation time curves, this is due to the facts that are explained in Section 6.3. This includes the redundant buffer control iterations on received messages, the Monitor with iterations that occur once in 5 simulation seconds and the Standard CAN bus implementation.
Additionally, TLS requires more events than the other protocols per additional ECU, as a Handshake is more costly than the TESLA or LASAN setup and more ECUs mean more receivers. Thus, the slope of this curve is steepest. TESLA starts at a higher computation time than LASAN as key generation is the process that requires the longest time for any ECU amount.

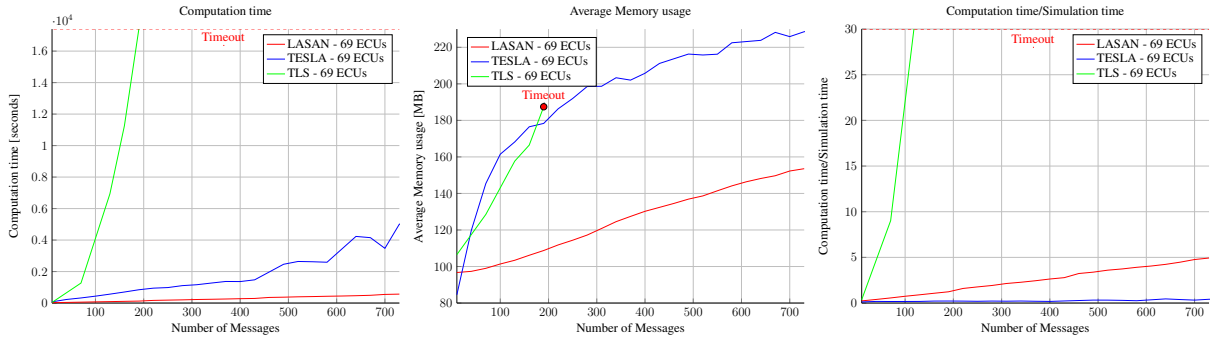

Figure 8.10: Standard mode. Computation time, average memory usage and ratio of computation time to simulation time plotted against number of messages.

The ratio of computation to simulation time behaves similarly as in the Rapid mode if ECUs are added. The difference of the slopes results again from the increased computation time in Standard mode. In contrast to the Rapid mode, the curve in TLS however, is linear in this case. This is because of the earlier explained steep increase of the computation time that dominates over the increase in simulation time that is similar to the Rapid mode.
The average memory behavior is similar to the Rapid mode for TLS and TESLA. This is because additional ECUs lead to more receivers that need to store messages for TLS. Thus, the TLS curve is going up linearly. In LASAN the average memory usage remains the same, similar to the Rapid mode. For LASAN the curve is increasing. This is due to the additional Monitor information that results from more ECUs and thus, more receivers in the system. This is dominating in the LASAN case as much Monitor information is given. On top of that, in LASAN the security module has to store more incoming messages in its receiving buffer, if additional ECUs are added that need to be registered.

Increase number of message streams

Results: If messages are added, for LASAN and TESLA, the computation time is rising linearly, which is similar to the Rapid mode case. In TLS it is going up with increasing slope. For 10 messages LASAN needs 23.69 s while for 730 messages it needs 566.85 s. For the same range, TESLA requires between 77.77 s and 5012.02 s. Values for TLS are out of range.
The ratio of computation time to simulation time behaves similar to the Rapid mode case. However, the slopes for the three curves are higher. For 10 to 730 messages for LASAN ratios between 0.24 and 4.93 are achieved and for TESLA between 0.12 and 0.41 are measured.
The average memory usage is linear for TLS and LASAN, which is similar to the Rapid mode. The curve for TESLA, however, is rising with a high slope that decreases as messages are added. For 10 to 730 messages, LASAN is in the range of 96.72 MB and 153.55 MB, TESLA is between 84.56 MB and 228.55 MB.

Interpretation and comparison: Here again the curves in Figure 8.10 show similar behavior as in Rapid mode. However, the slopes and magnitudes of the Standard curves for the consid-

ered performance indicators are higher if messages are added.
For the computation time curves, this is due to the same reasons as in the case when ECUs are added. TLS requires more events than the other two protocols on each additional message. Hence, as Handshakes are more costly than the TESLA or LASAN setup, this curve rises steepest. Because of the dominating key generation time, for TESLA a higher computation time value than LASAN is required independent of the number of sent message streams.
The average memory behavior is similar for TLS and LASAN if messages are added. However, the slope of those curves is steeper. This is due to additional Monitor information that needs to be stored for each message stream that is added to the system. If messages are added, TESLA changes from a linear behavior in the Rapid mode to a curve with decreasing slope for the Standard mode. This is because additional messages require more Monitor information to be stored. In TESLA this means that if there are more messages sent, the verification buffer is filled. For less sent messages, the information that is in this buffer is processed fast and thus, is not saved for a long time, not causing the process to allocate memory as other messages are removed from the buffer before new ones arrive. However, if more messages are sent, highly likely more messages are gathered in the verification buffer. This results in a higher number of messages that is stored in it at the same time. In the worst case this means that a receiver that receives 730 message streams stores one of each messages in the verification buffer at a time.
Similar to the case if ECUs are added, the ratio of computation to simulation time is similar to Rapid mode. The slope difference is present, because for additional messages, the computation time increases as explained above, while the additional simulation time per added message stream is small. In TLS the curve is linear as again the computation time dominates the simulation time.

8.2.4 Case-Study-Battery Management Performance

Experiment: To evaluate the performance of the simulator when integrated to the CPCSF, as presented in Chapter 7, the following experiment is performed. The CPCSF is started without GUI. Next, measurements are started with one bus in Rapid mode for a range of 5 to 60 cells in steps of 10 and are stopped if the cells finished their balancing. This is the case, if the difference between the highest SoC value and the lowest SoC value in the battery pack is smaller than 0.5 per cent. The cells start with a SoC distribution that is within a range of 3 per cent to around 60 per cent. SoC Broadcast messages are sent in an interval of 10 s. It is balanced with the Min-Max strategy, where the cell with highest SoC transmits its charge to the cell with lowest voltage until the voltages of all cells are within a specified range. For this experiment, the LASAN protocol is applied. This is important, as depending on the chosen strategy the simulation time and the computation time varies.

Results: The results of this experiment are illustrated in Figure 8.11. The execution time of the simulation increases with an increasing slope if the battery management simulation is started with additional cells. This ranges from 1.39 s that the simulation takes if 2 battery cells are used, to 19112.29 s if 60 batteries are used. The average memory usage goes up with an

Figure 8.11: BMS simulator connected to the IVNS in Rapid mode. Computation time, average memory usage and ratio of computation time to simulation time against number of battery cells in the system.

increasing slope when cells are added. Thus, for 2 cells, 95.68 MB are required, while for 60 cells 1344.69 MB are needed. The ratio of computation time to simulation time also increases with an increasing slope. Its values are between 0.0159 and 2.054 for the same cell range.

Interpretation: If more battery cells are added to the system, more messages are exchanged in order to finish the balancing. Each added battery cell adds SoC Broadcast and Status Response message, resulting in new events in the simulation on the sending and receiving side. With each new cell the number of receivers increases for all message types. Thus, as most messages are sent via broadcast, this leads to additional receiving events at all receivers for all types of messages that are exchanged. Moreover, if more cells are involved in the network, also more charge needs to be exchanged in order to get all cells balanced. This leads to a longer computation time and more transmitted messages in the simulation. Those factors result in a longer execution time, as longer simulation time and additional events, that come into play when the number of battery cells is extended, result in more processing steps. The average memory usage rises, if more cells are present in the simulation, as the BMS caches the resulting SoC values for the output. Lastly, the ratio of computation time to simulation time increases with additional battery cells. This is due to the fact that the simulation time itself is growing with a slightly decreasing slope that varies between 87 s for 2 cells and 9300.98 s for 60 cells. Thus, as the computation time rises and the simulation time decreases, the ratio increases.

In the first part of this chapter measurements of cryptographic operation times on a STM32 microcontroller are described and interpreted. Next, in the second part the performance of the proposed simulator is evaluated in terms of execution time and memory usage. This is done by running the test case generator that is introduced in Section 6.1 with various configurations. Then, the same evaluation is performed for the IVNS when combined with the CPCSF as it is explained in Chapter 7. After all aspects of the simulator are described so far, in the next chapter a short conclusion is given and future work is presented.

9 Conclusion and Future Work

In this thesis a modular and extensible discrete-event simulator has been designed and implemented. With this simulator it is possible to analyze the timing and conceptual behavior of security protocols in intra-vehicular networks with various configurations. It enables the user to evaluate if timing constraints are met within a system, if security protocols are suited conceptually for the special requirements of automotive networks (e.g. broadcast) or to get a better understanding of the processes in those networks.
For this purpose it allows the implementation of user-defined ECUs, gateways, buses and protocols. Those components can be arranged in any architecture and do consist of further subcomponents that can be modified and exchanged individually. They are used to communicate with each other on a realistically behaving system.
Realistic behavior is achieved by defining processing times and by using provided security implementations of operations such as certification or encryption. Those processing times depend on settings, such as cryptographic algorithms that are applied in the simulation. That is why this simulator provides the possibility to define dependencies between those settings and time variables within the components via an extensible database. This database is filled with processing times for cryptographic operations that were measured on a STM32 microcontroller.
Additionally, an extensible Observer was implemented that allows to extract user-defined information from the simulation. Connected to this, optionally, an extensible plug-in based GUI can be joined, that enables a deeper insight into the system's conceptual and temporal behavior.
On this simulator, the three security protocols LASAN, TESLA and TLS were implemented. Those protocols were then used together with a test case generator to evaluate the simulator in terms of performance. Also, in order to demonstrate a realistic use-case for the simulator, it was integrated with the CPCSF presented in [Mee15].
It is shown, that the simulator allows to evaluate systems of more than 1000 messages and 100

ECUs. On top of that, this flexible implementation makes it possible to easily compare simulations with different security parameters and thus, to analyze and compare the timing and system behavior depending on those chosen security parameters.

Future work on this topic includes the following. On a conceptual level this simulator can be extended to also allow cryptoanalysis. Thus, for instance an attacking component can be added to the environment in order to verify and optimize implemented security protocols. Also, further protocols can be implemented in the simulator including for example the proposed protocol LiBrA-CAN [GMVV12] or existing computer network authentication protocols such as Kerberos [KN93]. In addition to that, other bus technologies such as FlexRay or Automotive Ethernets can be added to the system in order to verify security protocols in those networks. Lastly, a comparison to simulators presented in Section 3.2 can be performed in terms of performance.

A

Appendix A

ECU Authentication Times		
Param.	**Function Type**	**Arguments**
$t_{VerSecCer}$	Cert. verification	$p_{HshSecCer}$, $p_{EncAlgSecCer}$, $p_{EncKeySecCer}$, $p_{EncModSecCer}$, $p_{CAsSecCer}$, s_{SecCer}
$t_{GenECUKey}$	sym. key generation	$p_{AlgECUKey}$, $p_{ModECUKey}$, $p_{KeyECUKey}$
$t_{EncReg1}$	public encryption	$p_{EncAlgReg1}$, $p_{EncKeyReg1}$, $p_{EncModReg1}$, $s_{DecReg1}$
$t_{HshReg1}$	hashing	$p_{HshReg1}$, $s_{DecReg1}$
$t_{EncReg2}$	signing	$p_{EncAlgReg2}$, $p_{EncKeyReg2}$, $p_{EncModReg2}$, $s_{HshReg1}$
$t_{DecReg1}$	private decryption	$p_{EncAlgReg1}$, $p_{EncKeyReg1}$, $p_{EncModReg1}$, $s_{EncReg1}$
$t_{DecReg2}$	verification	$p_{EncAlgReg2}$, $p_{EncKeyReg2}$, $p_{EncModReg2}$, $s_{EncReg2}$
$t_{HshRegCmp}$	hashing	$p_{HshReg1}$, $s_{DecReg1}$
$t_{VerECUCer}$	Cert. verification	$p_{HshECUCer}$, $p_{EncAlgECUCer}$, $p_{EncKeyECUCer}$, $p_{EncModECUCer}$, $p_{CAsECUCer}$, s_{ECUCer}
t_{EncCon}	sym. encryption	$p_{AlgECUKey}$, $p_{ModECUKey}$, $p_{KeyECUKey}$, s_{DecCon}
t_{DecCon}	sym. decryption	$p_{AlgECUKey}$, $p_{ModECUKey}$, $p_{KeyECUKey}$, s_{EncCon}

Table A.1: ECU Authentication database function requests with parameters in Arguments.

Stream Authorization Times		
Param.	**Function Type**	**Arguments**
t_{EncReq}	sym. encryption	$p_{AlgECUKey}$, $p_{ModECUKey}$, $p_{KeyECUKey}$, s_{DecReq}
t_{DecReq}	sym. decryption	$p_{AlgECUKey}$, $p_{ModECUKey}$, $p_{KeyECUKey}$, s_{EncReq}
$t_{GenSesKey}$	sym. key generation	$p_{AlgSesKey}$, $p_{ModSesKey}$, $p_{KeySesKey}$
t_{EncGra}	sym. encryption	$p_{AlgECUKey}$, $p_{ModECUKey}$, $p_{KeyECUKey}$, s_{DecGra}
t_{EncDen}	sym. encryption	$p_{AlgECUKey}$, $p_{ModECUKey}$, $p_{KeyECUKey}$, s_{DecDen}
t_{DecGra}	sym. decryption	$p_{AlgECUKey}$, $p_{ModECUKey}$, $p_{KeyECUKey}$, s_{EncGra}
t_{DecDen}	sym. decryption	$p_{AlgECUKey}$, $p_{ModECUKey}$, $p_{KeyECUKey}$, s_{EncDen}
t_{EncSim}	sym. encryption	$p_{AlgSesKey}$, $p_{ModSesKey}$, $p_{KeySesKey}$, $s_{SimpDec}$
t_{DecSim}	sym. decryption	$p_{AlgSesKey}$, $p_{ModSesKey}$, $p_{KeySesKey}$, $s_{SimpEnc}$

Table A.2: Stream Authorization database function requests with parameters in Arguments.

TESLA Times		
Param.	**Function Type**	**Arguments**
$t_{GenMACKey}$	Pseudo-Random-Function	$p_{MACAlgCre}$, $p_{MACKeyCre}$
$t_{EncKeyExc}$	asym/sym. encryption	$p_{AlgKeyExc}$, $p_{ModKeyExc}$, $p_{KeyKeyExc}$, $s_{DecKeyExc}$
$t_{DecKeyExc}$	asym/sym. decryption	$p_{AlgKeyExc}$, $p_{ModKeyExc}$, $p_{KeyKeyExc}$, $s_{EncKeyExc}$
t_{MACSim}	MAC generation	$p_{MACAlgLeg}$, $p_{MACKeyLeg}$, s_{DecSim}
$t_{PRFKeyDer}$	Pseudo-Random-Function	$p_{PRFAlgLeg}$
$t_{MACBufVer}$	MAC generation	$p_{MACAlgLeg}$, $p_{MACKeyLeg}$, s_{DecBuf}

Table A.3: TESLA database function requests with parameters in Arguments.

Record Layer Times		
Param.	**Function Type**	**Arguments**
$t_{MACCprRec}$	MAC generation	$p_{AlgRecMAC}$, $p_{KeyRecMAC}$, s_{CprRec}
t_{EncRec}	sym. encryption	$p_{AlgRecWri}$, $p_{KeyRecWri}$, $p_{ModRecWri}$, s_{CprRec}
t_{DecRec}	sym. decryption	$p_{AlgRecRea}$, $p_{KeyRecRea}$, $p_{ModRecRea}$, s_{EncRec}
$t_{MACVerCpr}$	MAC generation	$p_{AlgRecMAC}$, $p_{KeyRecMAC}$, $s_{MACCprRec}$

Table A.4: TLS Record Layer database function requests with parameters in Arguments.

Handshake Times		
Param.	**Function Type**	**Arguments**
$t_{VerSerCer}$	Cert. verification	$p_{HshSerCer}$, $p_{EncAlgSerCer}$, $p_{EncKeySerCer}$, $p_{EncModSerCer}$, $p_{CAsSerCer}$, s_{SerCer}
$t_{EncPreMas}$	public encryption	$p_{EncAlgSerCer}$, $p_{EncKeySerCer}$, $p_{EncModSerCer}$, $s_{DecPreMas}$
$t_{PRFMasSec}$	Pseud.-Ran.-Function	$p_{PRFMasSec}$, $s_{PRFInpMas}$
$t_{EncCerVer}$	public encryption	$p_{EncAlgSerCer}$, $p_{EncKeySerCer}$, $p_{EncModSerCer}$, $s_{DecCerVer}$
$t_{HshCliCac}$	hashing	p_{HshCac}, $s_{DecCliCac}$
$t_{PRFCliCac}$	Pseud.-Ran.-Function	p_{PRFCac}, $s_{HshCliCac}$
$t_{VerCliCer}$	Cert. verification	$p_{HshCliCer}$,$p_{EncAlgCliCer}$, $p_{EncKeyCliCer}$, $p_{EncModCliCer}$, $p_{CAsCliCer}$, s_{CliCer}
$t_{DecPreMas}$	private decryption	$p_{EncAlgSerCer}$, $p_{EncKeySerCer}$, $p_{EncModSerCer}$, $s_{EncCerVer}$
$t_{DecCerVer}$	private decryption	$p_{EncAlgSerCer}$, $p_{EncKeySerCer}$, $p_{EncModSerCer}$, $s_{EncCliCach}$
$t_{HshSerCac}$	hashing	p_{HshCac}, $s_{DecSerCac}$
$t_{PRFSerCac}$	Pseud.-Ran.-Function	p_{PRFCac},$s_{HshSerCac}$

Table A.5: TLS Handshake database function requests with parameters in Arguments.

B

Appendix B

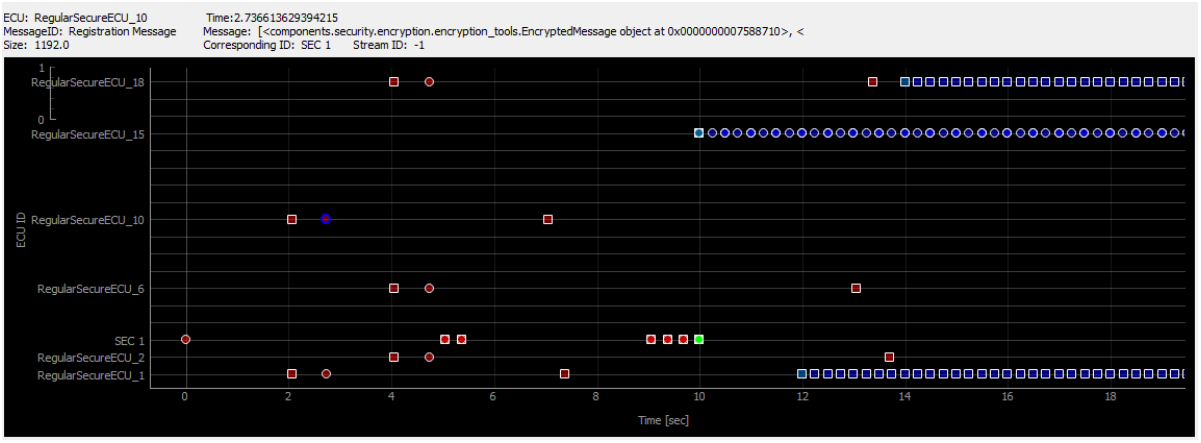

Figure B.1: Event View showing LASAN protocol. Circles resemble sent messages, while squares are received messages. Also red symbols indicate ECU Authentication message, green ones Stream Authorization and blue ones authorized messages.

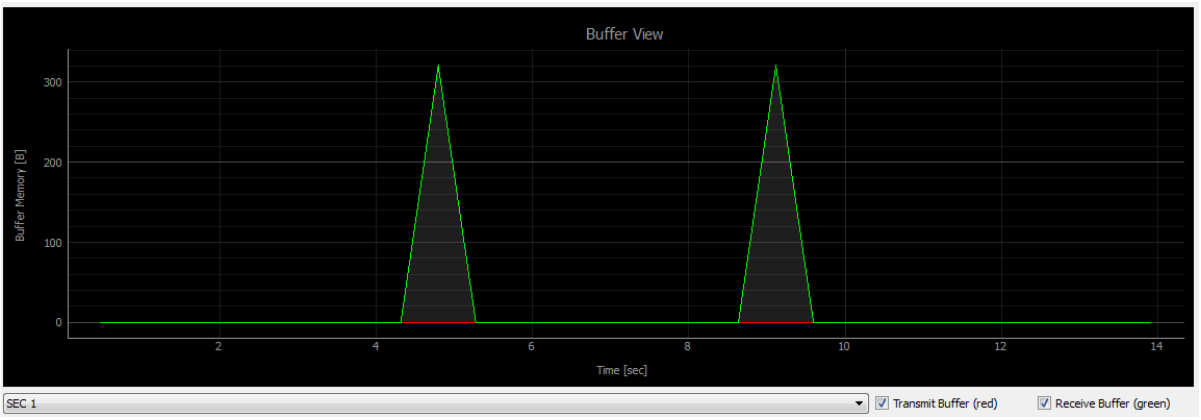

Figure B.2: Buffer View showing the buffer state of the security module of the LASAN protocol.

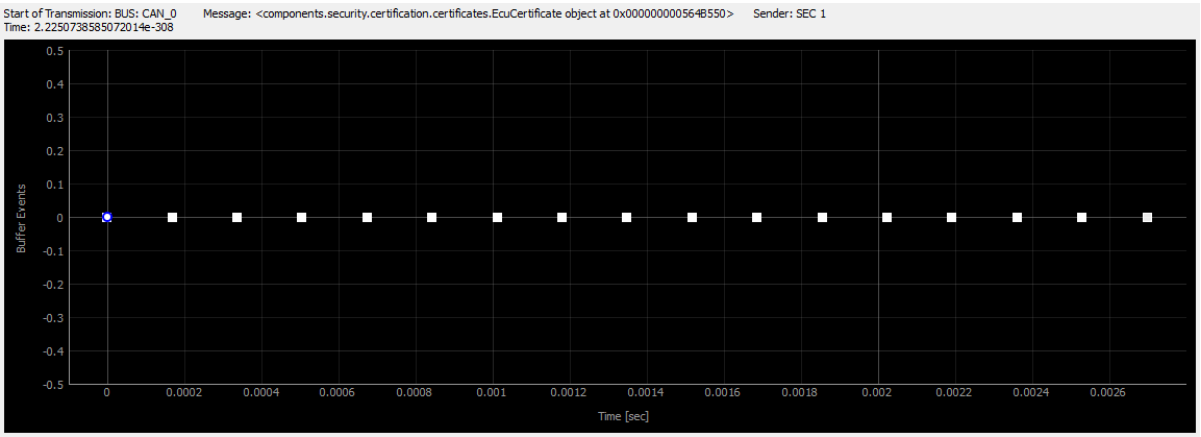

Figure B.3: CAN Bus State View shows individual frames that are transmitted on the buffer. By clicking the dots further information is displayed.

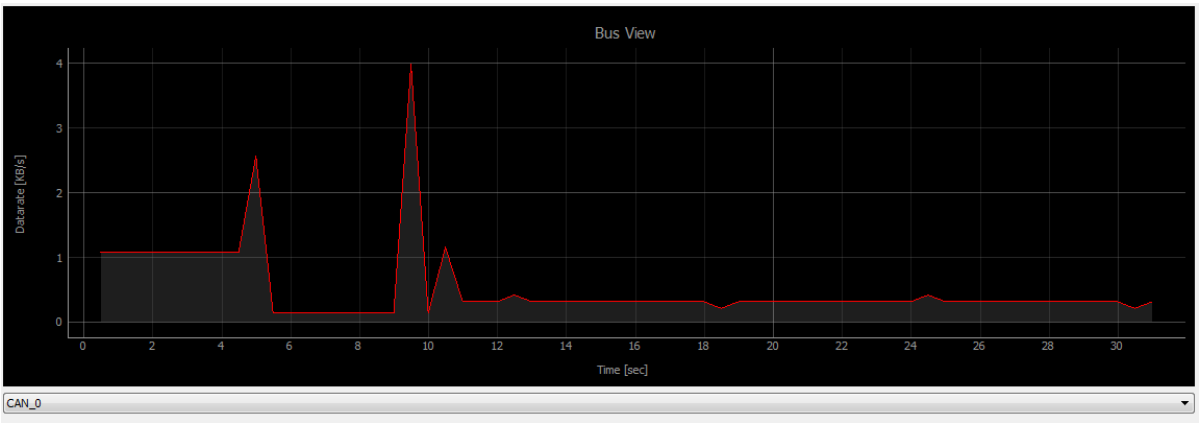

Figure B.4: Bus View shows the data rate of the selected CAN bus in time.

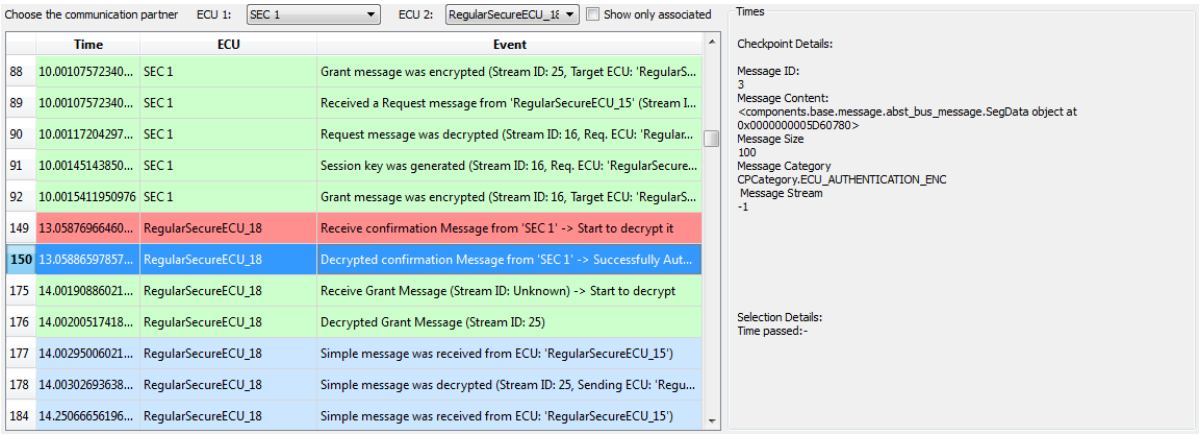

Figure B.5: Checkpoint View shows a list of all Monitor tags that are recorded during transmission. In this case LASAN is illustrated. Also red rows indicate ECU Authentication message, green ones Stream Authorization and blue ones authorized messages.

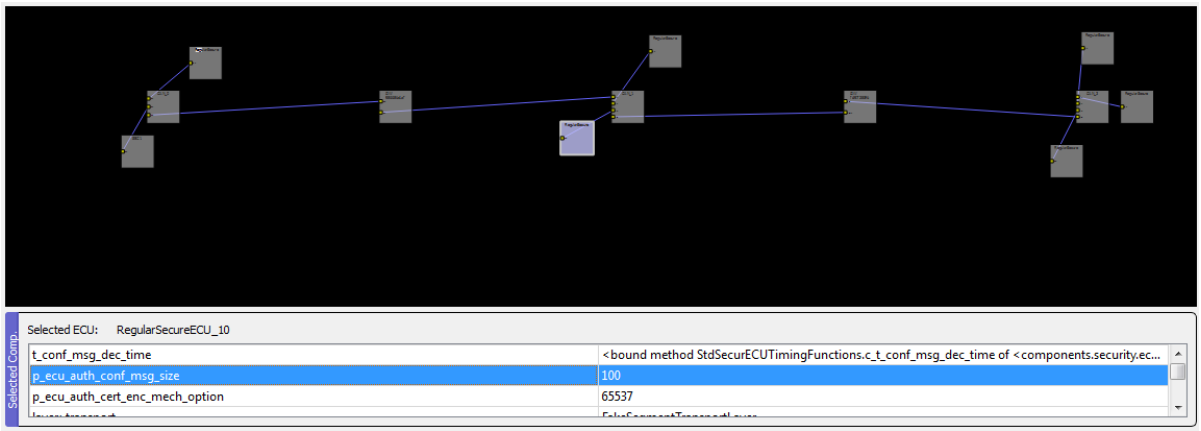

Figure B.6: Constellation View shows all components of the environment and their connections. By clicking on the individual component all settings of it are displayed.

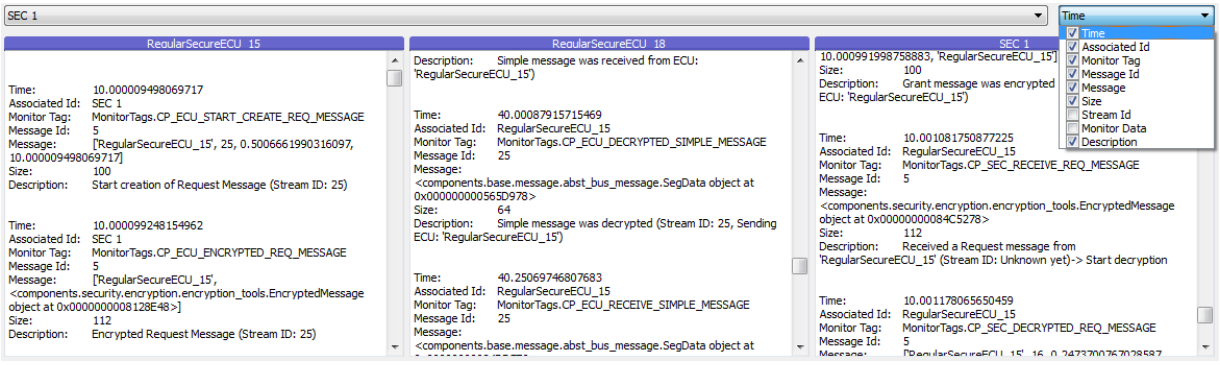

Figure B.7: ECU-Message View shows the incoming and outgoing events for each ECU individually. This output can be adjusted according to the individual components of a Monitor tag. E.g. displaying sizes or message content can be switched on or off.

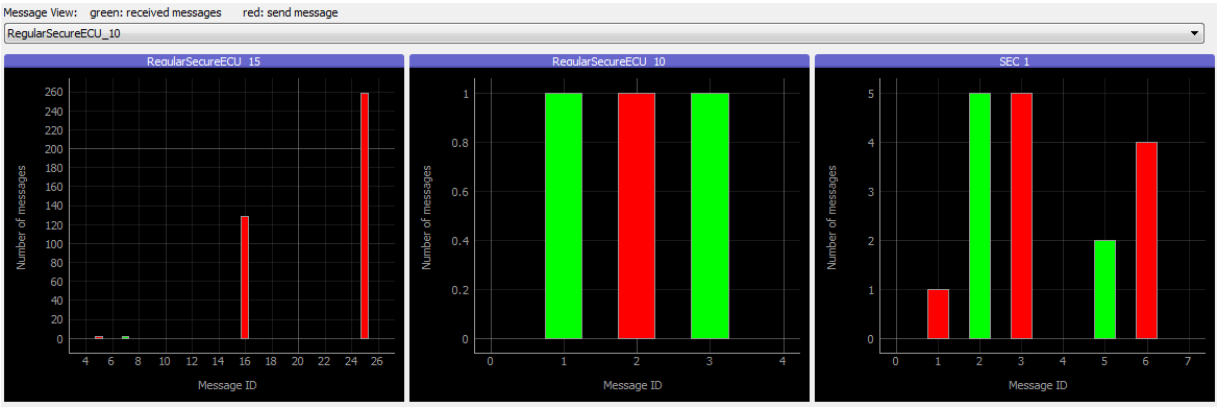

Figure B.8: Message-Count View shows the number of messages that are sent and received sorted by the message ID of those messages. Sent messages are green, while received ones are red.

Bibliography

[Aga15] AGARWAL, T.: Different Types of Microcontrollers that are used in Automobile Applications. (downloaded October 12th 2015). http://www.elprocus.com/different-microcontrollers-used-in-automobiles/

[App04] APPLIED DYNAMICS INTERNATIONAL: Real-time body module testing using SIMsystem and CANoe. (2004)

[Atm15] ATMEL CORPORATION: MegaAVR Microcontrollers. (downloaded October 12th 2015). http://www.atmel.com/products/microcontrollers/avr/megaAVR.aspx

[BBT04] BRIDAY, M. ; BECHENNEC, J.-L. ; TRINQUET, Y.: ReTiS: a real-time simulation tool for the analysis of distributed real-time applications. In: *Factory Communication Systems, 2004. Proceedings. 2004 IEEE International Workshop on*, 2004, S. 257–264

[Bec15] BECKER, D.: Wikipedia: ISO 15765-2. (downloaded October 22nd 2015). https://en.wikipedia.org/wiki/ISO_15765-2

[Bis05] BISHOP, M.: *Introduction to Computer Security*. Addison-Wesley, 2005. – ISBN 9780321247445

[Bis07] BISHOP, J.: *C# 3.0 Design Patterns*. O'Reilly Media, 2007. – ISBN 9780596551445

[BK12] BAJAJ, P. ; KHANAPURKAR, M.: Automotive networks based intra-vehicular communication applications. In: *New Advances in Vehicular Technology and Automotive Engineering, Prof. Joao Carmo (Ed.)*, 2012. – ISBN 9789535106982

[BNS05] BEUTELSPACHER, A. ; NEUMANN, H. B. ; SCHWARZPAUL, T.: *Kryptografie in Theorie und Praxis: mathematische Grundlagen für elektronisches Geld, Internetsicherheit und Mobilfunk*. Vieweg, 2005. – ISBN 9783528031688

[CHL+03] CERVIN, A. ; HENRIKSSON, D. ; LINCOLN, B. ; EKER, J. ; ÅRZÉN, K.-E.: How Does Control Timing Affect Performance? Analysis and Simulation of Timing Using Jitterbug and TrueTime. In: *IEEE Control Systems Magazine* 23 (2003), Nr. 3, S. 16–30

[CMK+11] CHECKOWAY, S. ; MCCOY, D. ; KANTOR, B. ; ANDERSON, D. ; SHACHAM, H. ; SAVAGE, S. ; KOSCHER, K. ; CZESKIS, A. ; ROESNER, F. ; KOHNO, T.: Comprehensive Experimental Analyses of Automotive Attack Surfaces. In: *USENIX Security Symposium* San Francisco, 2011

[CMLL13] CHOI, H.-H. ; MOON, J.-M. ; LEE, I.H. ; LEE, H.: Carrier Sense Multiple Access with Collision Resolution. In: *Communications Letters, IEEE* 17 (2013), Nr. 6, S. 1284–1287. – ISSN 1089–7798

[Con11] CONCEPCION, M.: *Diagnostics Strategies of Modern Automotive Systems: Automotive sensor testing*. Automotive Diagnostics and Pub., 2011. – ISBN 9781463552466

[DKM10] DEFO, G. B. ; KUZNIK, C. ; MULLER, W.: Verification of a CAN bus model in SystemC with functional coverage. In: *International Symposium on Industrial Embedded System (SIES)* (2010), S. 28–35. ISBN 9781424458394

[DR08] DIERKS, T. ; RESCORLA, E.: *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246 (Informational). `http://tools.ietf.org/html/rfc5246`. Version: August 2008 (Request for Comments)

[DSL+15] DARPA ; S. KESHAV CORNELL UNIVERSITY ; LBL ; XEROX PARC ; UCB ; USC/ISI ; SAMAN ; CONSER ; ACIRI: The Network Simulator - ns-2. (downloaded October 15th 2015). `http://www.isi.edu/nsnam/ns/`

[DZ83] DAY, J.D. ; ZIMMERMANN, H.: The OSI reference model. In: *Proceedings of the IEEE* 71 (1983), Nr. 12, S. 1334–1340. – ISSN 0018–9219

[Eck08] ECKERT, C.: *IT-Sicherheit: Konzepte, Verfahren, Protokolle*. Oldenbourg, 2008. – ISBN 9783486582703

[EJ01] EASTLAKE 3RD, D. ; JONES, P.: *US Secure Hash Algorithm 1 (SHA1)*. RFC 3174 (Informational). `http://www.ietf.org/rfc/rfc3174.txt`. Version: September 2001 (Request for Comments). – Updated by RFC 4634

[FDC11] FRANCILLON, A. ; DANEV, B. ; CAPKUN, S.: Relay Attacks on Passive Keyless Entry and Start Systems in Modern Cars. In: *Network and Distributed System Security Symposium* (2011), S. 431–439

[Fle05] FLEXRAY CONSORTIUM: FlexRay Communications System : Protocol Specification V2.1A. (2005)

[Fuj99] FUJIMOTO, R.M.: *Parallel and distributed simulation systems*. 1999. – 147–157 S. – ISBN 0780373073

[Ged15] GEDMINAS, M.: Python Object Graphs. (downloaded October 12th 2015). `https://mg.pov.lt/objgraph/#support-and-development`

[GMVV12] GROZA, B. ; MURVAY, S. ; VAN HERREWEGE, A. ; VERBAUWHEDE, I.: LiBrA-CAN: A lightweight broadcast authentication protocol for Controller Area Networks. In: *Proceedings of the International Conference on Cryptology and Network Security (CANS)* 7712 LNCS (2012), S. 185–200. – ISBN 9783642354038

[GZD+00] GAJSKI, D. ; ZHU, J. ; DÖMER, R. ; GERSTLAUER, A. ; ZHAO, S.: SpecC: Specification Language and Design Methodology, Kluwer Academic Publishers, 2000

[Har12] HARTWICH, F.: CAN with Flexible Data-Rate. In: *Proceedings of the 13th international CAN Conference* (2012)

[HWG11] HAO, J. ; WU, J. ; GUO, C.: Modeling and simulation of CAN network based on OPNET. In: *2011 IEEE 3rd International Conference on Communication Software and Networks, ICCSN 2011* (2011), S. 577–581. ISBN 9781612844855

[IAR15] IAR SYSTEMS GROUP: IAR Embedded Workbench. (downloaded October 12th 2015). `https://www.iar.com/iar-embedded-workbench/`

[Int11] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: ISO 15765-2: Road vehicles – Diagnostic communication over Controller Area Network (DoCAN) – Part 2: Transport protocol and network layer services. (2011)

[IXI14] IXIA: Automotive Ethernet : An Overview. (2014), Nr. May, S. 1–20. ISBN 9781107057289

[KCR+10] KOSCHER, K. ; CZESKIS, A. ; ROESNER, F. ; PATEL, S. ; KOHNO, T. ; CHECKOWAY, S. ; MCCOY, D. ; KANTOR, B. ; ANDERSON, D. ; SNACHÁM, H. ; SAVAGE, S.: Experimental security analysis of a modern automobile. In: *Proceedings - IEEE Symposium on Security and Privacy* (2010), S. 447–462. – ISBN 9780769540351

[KKM11] KOBLITZ, A.H. ; KOBLITZ, N. ; MENEZES, A.: Elliptic curve cryptography: The serpentine course of a paradigm shift. In: *Journal of Number theory* 131 (2011), Nr. 5, S. 781–814

[KN93] KOHL, J. ; NEUMAN, C.: *The Kerberos Network Authentication Service (V5)*. RFC 1510 (Proposed Standard). `http://www.ietf.org/rfc/rfc1510.txt`. Version: September 1993 (Request for Comments). – Obsoleted by RFC 4120

[KP88] KRASNER, G.E. ; POPE, S.T.: A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. In: *Journal of Object-Oriented Programming* 1 (1988), Nr. 3, S. 26–49

[LHD99] LEEN, G. ; HEFFERNAN, D. ; DUNNE, A.: Digital networks in the automotive vehicle. In: *Computing Control Engineering Journal* 10 (1999), Dec, Nr. 6, S. 257–266. – ISSN 0956–3385

[LIN10] LIN CONSORTIUM: LIN Specification Package, Revision 2.2A. (2010)

[LWWH11] LI, Z. ; WANG, T. ; WANG, Q. ; HOU, S.: Simulation of Acceleration Slip Regulation for vehicle based on CAN bus. In: *Proceedings 2011 International Conference on Transportation, Mechanical, and Electrical Engineering, TMEE 2011* (2011), S. 825–828. ISBN 9781457717017

[Mee15] MEEUW, A.: Master Thesis: Design, Implementation and Analysis of Distributed Communication for Smart Cell Batteries. (2015)

[Men15] MENTOR GRAPHICS CORPORATION: SystemVision CAN SI. (downloaded October 12th 2015). `https://www.mentor.com/products/sm/sv-can-network-si/`

[MfSDA14] MARTINS-FILHO, L.S. ; SANTANA, A. C. ; DUARTE, R.O. ; ARANTES JUNIOR, G.: Processor-in-the-Loop Simulations Applied to the Design and Evaluation of a Satellite Attitude Control. In: *Computational and Numerical Simulations* 1 (2014)

[MMT+13] MATSUMURA, J. ; MATSUBARA, Y. ; TAKADA, H. ; OI, M. ; TOYOSHIMA, M. ; IWAI, A.: A Simulation Environment based on OMNeT++ for Automotive CAN–Ethernet Networks. In: *Analysis Tools and Methodologies for Embedded and Real-time Systems* (2013), S. 1

[MNB11] MONOT, A. ; NAVET, N. ; BAVOUX, B.: Impact of clock drifts on CAN frame response time distributions. In: *Emerging Technologies Factory Automation (ETFA), 2011 IEEE 16th Conference on* (2011), S. 1–4. – ISBN 9781457700187

[MOS05] MOST COOPERATION: MOST Specification Rev 2.4 . In: *Most* (2005)

[MSL+15] MUNDHENK, P. ; STEINHORST, S. ; LUKASIEWYCZ, M. ; FAHMY, S.A. ; CHAKRABORTY, S.: Lightweight Authentication for Secure Automotive Networks. In: *Proceedings of the 2015 Design, Automation and Test in Europe Conference and Exhibition. EDA Consortium* (EDA Consortium, 2015), S. 285–288

[MV13] MILLER, C. ; VALASEK, C.: Adventures in automotive networks and control units. In: *Defcon 21* (2013)

[NGM09] NGUYEN, T. ; GUERSOY, M. ; METZNER, D.: System level modeling, simulation and verification of a CAN physical layer start network using behavior modeling language. In: *ECMS 2009 Proceedings edited by J. Otamendi, A. Bargiela, J. L. Montes, L. M. Doncel Pedrera* 6 (2009), Nr. Cd, S. 649–655. ISBN 9780955301889

[NIS01] NIST: Announcing the advanced encryption standard (AES), FIPS Pub. 197. In: *National Institute of Standards and Technology, Washington DC* (2001), S. 8–12. – ISSN 13534858

[NK14] NOH, D.H. ; KIM, D.S.: Message Scheduling on CAN Bus for Large-Scaled Ship Engine Systems. In: *World Congress* 19 (2014), Nr. 1, S. 7911–7916

[NS-15] NS-3 CONSORTIUM: The Network Simulator - NS-3. (downloaded October 21st 2015). https://www.nsnam.org/overview/what-is-ns-3/

[Ope15] OPENSIM LIMITED: OMNet++ - User Manual. (downloaded October 15th 2015). https://omnetpp.org/doc/omnetpp/manual/usman.html#sec100

[Ped15] PEDREGOSA, F.: Python Memory Profiler. (downloaded October 12th 2015). https://pypi.python.org/pypi/memory_profiler

[PSC+05] PERRIG, A. ; SONG, D. ; CANETTI, R. ; TYGAR, J. D. ; BRISCOE, B.: *Timed Efficient Stream Loss-Tolerant Authentication (TESLA): Multicast Source Authentication Transform Introduction*. RFC 4082 (Informational). http://www.ietf.org/rfc/rfc4082.txt. Version: June 2005 (Request for Comments)

[Rea15] REALTIME-AT-WORK: RTaW-Sim : Controller Area Network simulation and configuration. (downloaded October 12th 2015). http://www.realtimeatwork.com/software/rtaw-sim/

[Riv92] RIVEST, R.: *The MD5 Message-Digest Algorithm*. United States, 1992

[Riv15] RIVERBED TECHNOLOGY: Riverbed Modeler. (downloaded October 12th 2015). http://www.riverbed.com/products/steelcentral/steelcentral-riverbed-modeler.html

[RMM+10] ROUF, I. ; MILLER, R. ; MUSTAFA, H. ; TAYLOR, T. ; OH, S. ; XU, W. ; GRUTESE, M. ; TRAPPE, W. ; SESKAR, I.: Security and privacy vulnerabilities of in-car wireless networks: A tire pressure monitoring system case study. In: *Proceedings of the USENIX Security Symposium* 39 (2010), S. 11–13. – ISBN 8887666655554

[Rob91] ROBERT BOSCH GMBH: CAN Specification Version 2.0. (1991)

[RSA78] RIVEST, R.L. ; SHAMIR, A. ; ADLEMAN, L.: A method for obtaining digital signatures and public-key cryptosystems. In: *Communications of the ACM* 21 (1978), S. 120–126. – ISBN 00010782

[Sac92] SACHS, W.: *For Love of the Automobile: Looking Back Into the History of Our Desires*. University of California Press, 1992. – ISBN 9780520068780

[SD05] SCHIRNER, G. ; DÖMER, R.: Abstract communication modeling: A Case Study Using the CAN Automotive Bus. In: *Proceedings of International Embedded Systems Symposium* (2005)

[SKM+15] STEINHORST, S. ; KAUER, M. ; MEEUW, A. ; NARAYANASWAMY, S. ; LUKASIEWYCZ, M. ; CHAKRABORTY, S.: Cyber-Physical Co-Simulation Framework for Smart Cells in Scalable Battery Packs. (2015)

[SLN+14] STEINHORST, S. ; LUKASIEWYCZ, M. ; NARAYANASWAMY, S. ; KAUER, M. ; CHAKRABORTY, S.: Smart Cells for Embedded Battery Management. In: *Cyber-Physical Systems, Networks, and Applications (CPSNA), 2014 IEEE International Conference on*, 2014, S. 59–64

[SNA+13a] STUDNIA, I. ; NICOMETTE, V. ; ALATA, E. ; DESWARTE, Y. ; KAANICHE, M. ; LAAROUCHI, Y.: Security of embedded automotive networks: state of the art and a research proposal. In: *SAFECOMP 2013 - Workshop CARS (2nd Workshop on Critical Automotive applications : Robustness and Safety) of the 32nd International Conference on Computer Safety, Reliability and Security* (2013)

[SNA+13b] STUDNIA, I. ; NICOMETTE, V. ; ALATA, E. ; DESWARTE, Y. ; KAANICHE, M. ; LAAROUCHI, Y.: Survey on security threats and protection mechanisms in embedded automotive networks. In: *Proceedings of the International Conference on Dependable Systems and Networks* (2013), S. 1–12. – ISBN 9781479901814

[Som11] SOMMERVILLE, I.: *Software Engineering*. Pearson Education, 2011. – ISBN 9780133001495

[ST 13] ST MICROELECTRONICS LIMITED: STM32 Cryptographic Library - User manual. (2013)

[ST 15] ST MICROELECTRONICS LIMITED: STM32F415xx-Datasheet. (2015)

[Tea15] TEAM SIMPY: SimPy. (downloaded October 12th 2015). `https://simpy.readthedocs.org/en/latest/`

[TGH+15] TUOHY, S. ; GLAVIN, M. ; HUGHES, C. ; JONES, E. ; TRIVEDI, M. ; KILMARTIN, L.: Intra-vehicle networks: A review. In: *Intelligent Transportation Systems, IEEE Transactions on* 16 (2015), Nr. 2, S. 534–545

[The15] THE QT COMPANY LIMITED: Qt Documentation - Signal Slot. (downloaded October 12th 2015). `http://doc.qt.io/qt-4.8/signalsandslots.html`

[TW11] TANENBAUM, A.S. ; WETHERALL, D.: *Computer Networks*. Pearson Prentice Hall, 2011. – ISBN 9780132126953

[Vec15] VECTOR INFORMATIK GMBH: CANoe Homepage. (downloaded October 12th 2015). `http://vector.com/vi_canoe_en.html`

[Vos05] VOSS, W.: *A Comprehensible Guide to Controller Area Network*. Copperhill Technologies Corporation, 2005. – ISBN 9780976511601

[WH13] WANG, Y. ; HE, L.: Analysis and simulation of networked control systems delay characteristics based on TrueTime. 17 (2013), Nr. 4, S. 210–216

[Wol14] WOLFSSL: CyaSSL - User Manual. (2014), S. 1–320

[Wol15a] WOLFSSL: WolfSSL Benchmarks. (downloaded October 12th 2015). https://www.wolfssl.com/wolfSSL/benchmarks-wolfssl.html

[Wol15b] WOLFSSL: WolfSSL Homepage. (downloaded October 12th 2015). URL:https://www.wolfssl.com/wolfSSL/Home.html

[WWP04] WOLF, M. ; WEIMERSKIRCH, A. ; PAAR, C.: Security in Automotive Bus Systems. In: *Proceedings of the Workshop on Embedded Security in Cars* (2004), S. 1–13

[WWW07] WOLF, M. ; WEIMERSKIRCH, A. ; WOLLINGER, T.: State of the Art: Embedding Security in Vehicles. In: *EURASIP Journal on Embedded Systems* 2007 (2007), S. 1–16. – ISSN 1687–3955

[XMO13] XMOS LIMITED: XS1-L16A-128-QF124-Datasheet. (2013)

[ZLL+10] ZHANG, J. ; LEI, Y. ; LIU, H. ; ZHANG, X. ; LIU, Z.: Application of CANoe-Matlab Co-simulation in DCT-CAN Bus Control. In: *Automobile Technology* 9 (2010)

[ZS11] ZIMMERMANN, W. ; SCHMIDGALL, R.: *Bussysteme in der Fahrzeugtechnik*. Vieweg+Teubner, 2011. – ISBN 9783834802354

[ZTS11] ZIERMANN, T. ; TEICH, J. ; SALCIC, Z.: DynOAA Dynamic offset adaptation algorithm for improving response times of CAN systems. In: *2011 Design Automation Test in Europe* (2011), S. 1–4. – ISBN 9783981080179